Subtyping Delimited Continuations

Marek Materzok
Institute of Computer Science
University of Wrocław
Wrocław, Poland
tilk@tilk.eu

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

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 arbitrary contexts 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 keeps pure expressions in direct style.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs

General Terms Languages, Theory

Keywords Delimited Continuation, Continuation-Passing Style, Type System, Subtyping

1. Introduction

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 *call/cc* present in Scheme [23] and SML/NJ [20], delimited-control operators model composition of delimited (partial) continuations rather than jumps to undelimited continuations. In particular, while *call/cc* offers the programmer the power of the continuation-passing style (CPS) in direct style, Danvy and Filinski's *shift* and *reset* offer the power of the continuation-composing style (CCS) 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 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 [18, 19]. A long list of non-trivial applications of delimited continuations includes normalization by evaluation and partial evaluation [3, 12, 15], mobile computing [33], linguistics [4], and operating systems [25] to name but a few.

The intuitive semantics of delimited-control operators such as *shift* ($\mathcal{S}$) and *reset* ($\langle\rangle$) can be explained as follows. When the expression $\mathcal{S}k.e$ is evaluated, the current—delimited by the nearest dynamically-enclosing reset—continuation is captured, i.e., it is removed and the variable k is bound to it. When later k is applied to a value, the then-current continuation is composed with the captured continuation. For example, the following expression (where ++ is string concatenation):

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

evaluates to the string *“Alice has a dog and the dog has a cat.”*

In their first article on delimited continuations [13], Danvy and Filinski briefly described a variant of *shift* and *reset*, known as $shift_0$ ($\mathcal{S}_0$) and $reset_0$, where the operational semantics of the control delimiter $reset_0$ coincides with that of *reset*, but $shift_0$, when capturing a continuation, instead of resetting it, replaces it with the inner-most delimited continuation pending on the metacontinuation. In other words, in contrast to *shift*, $shift_0$ removes the control delimiter along with the captured continuation, which makes it possible to use $shift_0$ to access continuations arbitrarily deep in the metacontinuation by repeatedly shifting continuations. For example, the following expression:

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

evaluates to the string *“A cat has Alice.”*, since the second shifting has access to the continuation that prepends the string *“Alice”* to a given string. The ability to access arbitrary contexts (a first-order representation of continuations [9]) in the metacontext (a first order representation of the metacontinuation [9])) makes $\mathcal{S}_0$ a particularly interesting and powerful control operator.

In the untyped setting, $shift_0$ and $reset_0$ have been investigated by Shan [32], who presented a CPS translation for $shift_0$ and $reset_0$ that conservatively extends Plotkin's call-by-value CPS translation [28] 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 [26] introduced a substructural type system for $shift_0$ and $reset_0$ that abstractly interprets

ICFP'11, September 19–21, 2011, Tokyo, Japan.

(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 which is the most expressive of the known monomorphic type systems for *shift* and *reset*. 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 $\langle 1 + \mathcal{S}k.\, \mathit{true}\rangle$ is well typed in this type system. The typing rules of Danvy and Filinski's type system have been derived from a left-to-right call-by-value evaluator in continuation-passing style [13].

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 a new CPS translation for *shift*$_0$ and *reset*$_0$ which conservatively extends Plotkin's call-by-value CPS translation and which turns out to be close to a curried version of Shan's CPS translation [32]. From this CPS translation, we derive an abstract machine and 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 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 [26], 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 in 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 λ-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 CPS translations for *shift*$_0$ and *reset*$_0$ that we present induce new CPS translations for *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:

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

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

According to the above rule for *shift*, the type system assigns the captured continuation a functional type with a single, chosen upfront 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 upfront and are subject to the same restrictions as described above.

In a way, one can treat the continuation types from [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 the means of subtyping of 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.

The rest of this article is structured as follows. In Section 2.1, we introduce the syntax and reduction semantics for the call-by-value λ-calculus with *shift*$_0$ and *reset*$_0$. In Section 2.2, we present a CPS translation for the language and we relate it to the reduction semantics. In Section 2.3, we derive a type-and-effect system à la Danvy and Filinski from the CPS. In Sections 3.1 and 3.2, we introduce subtyping to the type system and we prove several standard properties for the system, including strong type soundness. In Section 3.3, we use a context-based method of reducibility predicates to prove termination of evaluation of well-typed programs. In Section 3.4, we present and prove correct a type inference algorithm for our type system. In Section 3.5, 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*$_0$. In Sections 6 and 7, we discuss related and future work and we conclude in Section 8.

The article is accompanied by a Haskell implementation of the presented programming language and by a Twelf formalization of its metatheory, both available at `http://www.tilk.eu/shift0/`.

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

2.1 Syntax and semantics

In this subsection, 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*$_0$ ($\langle\langle\rangle\rangle$).

We introduce three syntactic categories of terms, evaluation contexts and trails (i.e., stacks of evaluation contexts):

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

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 inside a context:

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

$$
\begin{array}{rl}
\text{Environments:} & \rho ::= \rho_{\text{init}} \mid \rho[x \mapsto v] \\
\text{Values:} & v ::= [x, e, \rho] \mid k \\
\text{Contexts:} & k ::= \text{End} \mid \text{Arg}(e, \rho, k) \mid \text{Fun}(v, k) \\
\text{Metacontexts:} & mk ::= [\,] \mid k : mk \\
\\
\text{Initial configurations:} & \langle e, \rho_{\text{init}}, \text{End} : [\,] \rangle_{\text{eval}} \\
\text{Final configurations:} & \langle \text{End}, v, [\,] \rangle_{\text{cont}}
\end{array}
$$

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

Figure 1. Abstract machine for $\lambda^{\mathcal{S}_0}$

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

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

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

Figure 3. Untyped uncurried CPS translation for $\lambda^{\mathcal{S}_0}$

There are three contraction rules:

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

where $e[x/e']$ stands for the usual capture-avoiding substitution of e' 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 rule for *shift*

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

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. This feature has a big impact on the type systems that we present later in the article.

Finally, we define the reduction relation on terms representing programs (i.e., triples $\langle e, T, K \rangle$):

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

where any closed non-value term can be uniquely represented as $K[T[e]]$. 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.

2.2 CPS translation

The *shift*$_0$ control operator allows access to every context in the trail. What is more, unlike with *shift*, 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 [32], the trail is represented explicitly as a list of contexts which must 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 2. Evaluating a well-delimited (i.e., with no dangling *shift*$_0$'s) 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* $\lambda_{\to}$.

For reference, and to make it possible for the reader to compare the evaluation model encoded in our initial CPS translation with the existing ones [10, 16, 32], we briefly discuss possible modifications of the CPS translation. First of all, the translation can be modified so that every translated term takes all its continuation parameters in one list parameter. Figure 3 presents such an uncurried CPS translation that targets the untyped lambda-calculus extended with a separate syntactic category of lists and two new kinds of expressions: list abstraction λ^{l} and list application @. The translation underlies a continuation-passing style interpreter that, when defunctionalized [1, 29], gives rise to the abstract machine of Figure 1. This abstract machine can be seen to be equivalent with the abstract machine for *shift*$_0$/*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 [7, 8].) 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 [32] and which corresponds exactly to the abstract machine of Biernacki et al. [10].

2.3 From CPS translation to a type system

With the CPS translation defined, one can use the approach of Danvy and Filinski [13] and build a type system based on this 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

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

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

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

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

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:

types	τ	$::=$	$\alpha \mid \tau \xrightarrow{\sigma} \tau$
annotations	σ	$::=$	$\epsilon \mid [\tau\ \sigma]\ \tau\ \sigma$

The resulting type system for $\lambda^{\mathcal{S}_0}$ is shown in Figure 4.

The type annotations require some explanation. Their meaning is best understood together with types they annotate. The typing judgment $\Gamma \vdash e : \tau_1'\ [\tau_1\ \sigma_1]\ \tau_2'\ \ldots\ [\tau_n\ \sigma_n]\ \tau$ (we omit ϵ for brevity) 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 to be associated with the return type.

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

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

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 for clarity) to

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

This 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 k 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 $\lambda_\leq^{\mathcal{S}_0}$

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

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

All subtyping rules are rather straightforward, except the rule for $\epsilon \leq [\tau\ \sigma]\ \tau\ \sigma$, which seems non-intuitive. To get some understanding of what this rule means, let us consider an effect-annotated type where the contexts are effect-free: $\tau_1\ [\tau_1']\ \tau_2\ \ldots\ \tau_n\ [\tau_n']\ \tau_f$. For a pure τ to be a subtype of this type, it is required by the rule for $\epsilon \leq [\tau\ \sigma]\ \tau\ \sigma$ that $\tau_k' \leq \tau_{k+1}$ for all k; in other words, that the contexts can be composed.

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.

Theorem 3 (Subject reduction)**.** *If* $\vdash e : \tau\ \sigma$ *and* $e \to e'$ *for some* e'*, then* $\vdash e' : \tau\ \sigma$.

Theorem 4 (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* $\sigma = \epsilon$*, the third case cannot occur.*

Theorem 5 (Unique decomposition)**.** *If* $\vdash e : \tau$ *and* $e \to e'$ *for some* e'*, then* $e = K[T[r]]$ *for exactly one context* K*, trail* T *and redex* r.

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.

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

$$\frac{\tau \le \tau' \qquad \sigma \le \sigma'}{\tau\,\sigma \le \tau'\,\sigma'} \qquad \frac{}{\alpha \le \alpha} \qquad \frac{\tau_2' \le \tau_1' \qquad \tau_1\,\sigma_1 \le \tau_2\,\sigma_2}{\tau_1' \xrightarrow{\sigma_1} \tau_1 \le \tau_2' \xrightarrow{\sigma_2} \tau_2}$$

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

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

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

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

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

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 $(\pi'\,\pi)\tau\sigma = \pi'(\pi\tau\sigma)$.*

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 2 (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 3 (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 4 (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, we have

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

but it is not a valid typing for $\lambda x.x\,(\mathcal{S}_0 f.y)$.

However, every pure context is an impure one:

Lemma 5 (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 6 (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 7 (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:

Contexts

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

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

Pure contexts

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

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

Trails

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

Subtyping for trail types

$$\frac{}{\iota \le \iota} \qquad \frac{\tau_1 \xrightarrow{\sigma} \tau_2 \le \tau_1' \xrightarrow{\sigma'} \tau_2' \qquad \pi \le \pi'}{\tau_1 \xrightarrow{\sigma} \tau_2\ \pi \le \tau_1' \xrightarrow{\sigma'} \tau_2'\ \pi'}$$

Metacontexts, programs

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

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

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 12 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.

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

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

Lemma 8. *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 9. *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.
   - $\sigma' = \epsilon$, $\sigma = \epsilon$. 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).
   - $\sigma' = \epsilon, \sigma = [\tau_1\,\sigma_1]\,\tau_2\,\sigma_2$. 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 8. 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 10. *For every type τ, $\mathcal{T}_{\tau \to \tau}(\bullet)$ 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 $\mathcal{M}_\tau(T_M, K)$. 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 11. *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 12. *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, $\mathcal{N}(e\{\vec{x}/\vec{v}\}, T \cdot T_M, K)$ holds.*

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

- $D = \text{SUB}(D', \tau'\,\sigma' \leq \tau\,\sigma)$. From Lemma 9 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.
- $D = \text{VAR}$, $e = x_i$. So we have $e\{\vec{x}/\vec{v}\} = v_i$, $\tau = \tau_i$, $\sigma = \epsilon$,$T = \Box$, $\mathcal{M}_{\tau_i}(T_M, K)$. By assumption $\mathcal{R}_{\tau_i}(v_i)$ and the definition of $\mathcal{M}_{\tau_i}$ we have $\mathcal{N}(v_i, T_M, K)$.
- $D = \text{ABS}(D')$, $e = \lambda x.e'$. So $\tau = \tau'' \to \sigma'\tau'$, $\sigma = \epsilon$, $T = \Box$, $\mathcal{M}_\tau(T_M, K)$. As in the previous case, to show that $\mathcal{N}((\lambda x.e')\{\vec{x}/\vec{v}\}, T_M, K)$ we need $\mathcal{R}_\tau(\lambda x.e'\{\vec{x}/\vec{v}\})$.
  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 $\mathcal{N}((\lambda x.e'\{\vec{x}/\vec{v}\})\,v, T \cdot T_M, K)$. Because $(\lambda x.e'\{\vec{x}/\vec{v}\})\,v$ reduces in one step to $e'\{\vec{x}/\vec{v}\}\{x/v\}$, it suffices to show $\mathcal{N}(e'\{\vec{x}/\vec{v}\}\{x/v\}, T \cdot T_M, K)$, which follows from the induction hypothesis.
- $D = \text{APP}(D_1, D_2)$, $e = e_1\,e_2$. 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 $e'_1 = e_1\{\vec{x}/\vec{v}\}$ and $e'_2 = \{\vec{x}/\vec{v}\}$. 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 $\mathcal{M}_{\downarrow\tau_B\,\sigma_B}(T_{KR}, M_R)$. We have to show that $\mathcal{N}((\lambda x.\langle K_1[v_L\,x]\rangle)\,v_R, T_R \cdot T_{KR}, M_R)$. 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 $\mathcal{R}_{\tau'} \xrightarrow{[\tau_A\,\sigma_A]\,\tau_B\,\sigma_B}_{\tau}(v_L)$.
- $D = \text{APP-PURE}(D_1, D_2)$, $e = e_1\,e_2$. The proof is similar to the case for APP.
- $D = \text{SHIFT0}(D')$, $e = \mathcal{S}_0 f.e'$. So $\sigma = [\tau'\sigma']\tau''\sigma''$, $T = K_1 \cdot T'$, $\mathcal{R}_\tau \xrightarrow{\sigma'}_{\tau'}(\lambda x.\langle K_1[x]\rangle)$, $\mathcal{T}_{\overrightarrow{\tau'\sigma''}}(T')$ and $\mathcal{M}_{\downarrow\tau''\sigma''}(T_M, K)$. We want to show $\mathcal{N}(\mathcal{S}_0 f.e\{\vec{x}/\vec{v}\}, K_1 \cdot T' \cdot T_M, K)$, which is equivalent to $\mathcal{N}(\langle K_1[\mathcal{S}_0 f.e\{\vec{x}/\vec{v}\}]\rangle, T' \cdot T_M, K)$. Because $\langle K_1[\mathcal{S}_0 f.e\{\vec{x}/\vec{v}\}]\rangle$ reduces to $e\{\vec{x}/\vec{v}\}\{f/\lambda x.\langle K[x]\}$, it suffices to show that $\mathcal{N}(e\{\vec{x}/\vec{v}\}\{f/\lambda x.\langle K[x]\}, T' \cdot T_M, K)$, which follows from the induction hypothesis.
- $D = \text{RESET0}(D')$, $e = \langle e'\rangle$. Then for some τ' we have $\Gamma \vdash e' : \tau'\,[\tau']\,\tau\,\sigma$. From Lemma 10 have $\mathcal{T}_{\tau'\to\tau'}(\bullet)$, and thus $\mathcal{T}_{\overrightarrow{\tau'\,[\tau']\,\tau\,\sigma}}(\bullet \cdot T)$. The induction hypothesis gives us $\mathcal{N}(e'\{\vec{x}/\vec{v}\}, \bullet \cdot T \cdot T_M, K)$, which is equivalent to the thesis, $\mathcal{N}(\langle e'\rangle\{\vec{x}/\vec{v}\}, \cdot T \cdot T_M, K)$.

□

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 11 we have $\mathcal{M}_\tau(\Box, \bullet)$, thus the theorem follows from Lemma 12. □

3.4 Type inference

Type inference in $\lambda^{\mathcal{S}_0}_{\leq}$ is decidable. For the purpose of type inference, we replace the rules APP and APP-IND 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 $\ll$ is defined as follows:

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

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

Expressions of the form $\sigma \leq^? \sigma$ will be called *subtyping constraints*, and expressions of the form $\sigma \ll^? \vec{\sigma}$ will be called

$$
\begin{array}{rcl}
[\![\alpha]\!] & = & \alpha \\
[\![\tau_1 \xrightarrow{\sigma} \tau_2]\!] & = & [\![\tau_1]\!] \to [\![\tau_2\ \sigma]\!] \\
[\![\tau\ \epsilon]\!] & = & [\![\tau]\!] \\
[\![\tau\ [\tau_1\ \sigma_1]\ \tau_2\ \sigma_2]\!] & = & ([\![\tau]\!] \to [\![\tau_1\ \sigma_1]\!]) \to [\![\tau_2\ \sigma_2]\!] \\
[\![\alpha \le \alpha]\!] & = & \lambda x.x \\
[\![\tau_1' \xrightarrow{\sigma_1} \tau_1 \le \tau_2' \xrightarrow{\sigma_2} \tau_2]\!] & = & \lambda f.\lambda x.[\![\tau_1\ \sigma_1 \le \tau_2\ \sigma_2]\!](f([\![\tau_2' \le \tau_1']\!]x)) \\
[\![\tau\ \epsilon \le \tau'\ \epsilon]\!] & = & [\![\tau \le \tau']\!] \\
[\![\tau\ \epsilon \le \tau'\ [\tau_1\ \sigma_1]\ \tau_2\ \sigma_2]\!] & = & \lambda x.\lambda f.[\![\tau_1\ \sigma_1 \le \tau_2\ \sigma_2]\!](f([\![\tau \le \tau']\!]x)) \\
[\![\tau\ [\tau_1\ \sigma_1]\ \tau_1'\ \sigma_1' \le \tau'\ [\tau_2\ \sigma_2]\ \tau_2'\ \sigma_2']\!] & = & \lambda f.\lambda g.[\![\tau_1'\ \sigma_1' \le \tau_2'\ \sigma_2']\!](f(\lambda x.[\![\tau_2\ \sigma_2 \le \tau_1\ \sigma_1]\!](g([\![\tau \le \tau']\!]x)))) \\
[\![x]\!]_{\text{VAR}} & = & x \\
[\![e]\!]_{\text{SUB}(D,\tau\ \sigma \le \tau'\ \sigma')} & = & [\![\tau\ \sigma \le \tau'\ \sigma']\!][\![e]\!]_D \\
[\![\lambda x.e]\!]_{\text{ABS}(D)} & = & \lambda x.[\![e]\!]_D \\
[\![fe]\!]_{\text{APP-PURE}(D_1,D_2)} & = & [\![f]\!]_{D_1}[\![e]\!]_{D_2} \\
[\![fe]\!]_{\text{APP}(D_1,D_2)} & = & \lambda k.[\![f]\!]_{D_1}(\lambda f.[\![e]\!]_{D_2}(\lambda e.fek)) \\
[\![\mathcal{S}_0 f.e]\!]_{\text{SHIFT0}(D)} & = & \lambda f.[\![e]\!]_D \\
[\![\langle e \rangle]\!]_{\text{RESET0}(D)} & = & [\![e]\!]_D(\lambda x.x)
\end{array}
$$

Figure 7. Type-directed CPS translation for $\lambda_{\le}^{\mathcal{S}_0}$

composition 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 call a substitution σ-grounding, if its values do not contain any annotation variables. We say that a substitution s solves a constraint $\sigma_1 \le^? \sigma_2$ ($\sigma_1 \ll^? \vec{\sigma}$) if and only if $s(\sigma_1) \le s(\sigma_2)$ ($s(\sigma_1) \ll s(\vec{\sigma})$) is derivable.

Theorem 8 (Principal types)**.** *For every term e and typing environment Γ there exist: α, δ, a set of $\tau\sigma$-subtyping constraints S, and a set of composition constraints C, such that:*

1. *For every σ-grounding 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 σ-grounding substitution s such that $s(\alpha) = \tau$, $s(\delta) = \sigma$ and s solves S and C.*

The type inference algorithm finds the principal type (along with the unsolved constraints) in polynomial time and it also generates a typing derivation skeleton, where subtyping derivations have to be filled in. In order to check if the term is actually well-typed (and to obtain a concrete typing derivation), one has to find a σ-grounding substitution which solves the inequalities.

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 7, targets the simply-typed lambda calculus, and as the CPS translation of Figure 2, it preserves types:

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

The derivations of the subtyping relation are translated to coercion functions, which are typed as expected:

Lemma 13. *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 $\lambda_{\to}$.*

Another 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 [24], [27], and recently in [31], which also use effect annotations to distinguish pure terms from effectful ones. The property suggests that the translation presented here may be useful in implementing *shift*$_0$/*reset*$_0$ in functional programming languages.

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 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 $\lambda_{\le}^{\mathcal{S}_0}$.

4. Programming examples

4.1 Prototype implementation

In order to study the applications of the type system $\lambda_{\le}^{\mathcal{S}_0}$, 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:

$$
\begin{array}{rcl}
e & ::= & \ldots \mid \text{true} \mid \text{false} \mid \text{if } e \text{ then } e \text{ else } e \\
& & \mid \text{nil} \mid \text{cons}(e, e) \mid \text{case } e\ \{\text{nil} \to e; \text{cons}(x_1, x_2) \to e\} \\
& & \mid k \mid e \oplus e \mid e \gtrless e \mid \text{fix} x.e \\
\tau & ::= & \ldots \mid \text{int} \mid \text{bool} \mid \text{list}(\tau)
\end{array}
$$

The typing rules for new expressions introduced above are given in Figure 8.

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.

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

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

$$\frac{}{\Gamma \vdash k : \text{int}}\ \text{INT}$$

$$\frac{\Gamma \vdash k_1 : \text{int}\ \sigma_1 \quad \Gamma \vdash k_2 : \text{int}\ \sigma_2 \quad \sigma \ll \sigma_1\ \sigma_2}{\Gamma \vdash k_1 \oplus k_2 : \text{int}\ \sigma}\ \text{ARITH}$$

$$\frac{\Gamma \vdash k_1 : \text{int}\ \sigma_1 \quad \Gamma \vdash k_2 : \text{int}\ \sigma_2 \quad \sigma \ll \sigma_1\ \sigma_2}{\Gamma \vdash k_1 \gtreqless k_2 : \text{bool}\ \sigma}\ \text{REL}$$

$$\frac{}{\Gamma \vdash \text{nil} : \text{list}(\tau)}\ \text{NIL}$$

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

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

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

Figure 8. Typing rules for booleans, integers, lists and recursion

4.2 Example—*prefixes*

The *prefixes* example shown in Figure 9 has been considered in [2] and in [6]. The function `prefixes` lists prefixes of a given list, e.g., `prefixes [1,2,3]` yields `[[1],[1,2],[1,2,3]]`. 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 occurrence (`k []`) the continuation is simply given the pure type $\text{list(int)} \to \text{list(int)}$. In the second (`k (w xs)`), it is given a more complex type $\text{list(int)} \xrightarrow{[\text{list(int)}]\ \text{list(int)}} \text{list(int)}$.

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 10. 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:

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

Hence, it requires two contexts, both of the type $\text{list(int)} \to \text{list(int)}$, 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.

```
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>
```

Figure 9. 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>>
```

Figure 10. Example program—*partition*

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

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

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

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

Using the translation from Figure 7 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

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 *shift*/*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):

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

The translation on typing derivations is shown in Figure 11. 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* $\lambda^{\mathcal{S}_0}_{\le}$.

One can also consider the CPS translations in both systems $\lambda^{\mathcal{S}}_{\to}$ and $\lambda^{\mathcal{S}_0}_{\le}$. First, we notice that the CPS translations of the typing derivations resulting from the translation in Figure 11 are β-equal

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

$$\frac{\Gamma, x : \tau; \tau_1 \vdash e : \tau'; \tau_2}{\Gamma; \tau'' \vdash \lambda x.e : (\tau\ {}_{\tau_1} \to {}_{\tau_2}\ \tau'); \tau''}\ \text{ABS} \qquad \frac{\dfrac{\Gamma, x : \tau \vdash e : \tau'\ [\tau_1]\ \tau_2}{\Gamma \vdash \lambda x.e : \tau \xrightarrow{[\tau_1]\ \tau_2} \tau'}\ \text{ABS} \quad \dfrac{\dots}{\tau \xrightarrow{[\tau_1]\ \tau_2} \tau' \le \tau \xrightarrow{[\tau_1]\ \tau_2} \tau'[\tau'']\ \tau''}}{\Gamma \vdash \lambda x.e : \tau \xrightarrow{[\tau_1]\ \tau_2} \tau'\ [\tau'']\ \tau''}\ \text{SUB}$$

$$\frac{\Gamma; \tau_4' \vdash f : (\tau_1\ {}_{\tau_1'} \to {}_{\tau_3'}\ \tau_2); \tau_2' \quad \Gamma; \tau_3' \vdash e : \tau_2; \tau_4'}{\Gamma; \tau_1' \vdash fe : \tau_2; \tau_2'}\ \text{APP} \qquad \frac{\Gamma \vdash f : \tau_1 \xrightarrow{[\tau_1']\ \tau_3'} \tau_2\ [\tau_4']\ \tau_2' \qquad \Gamma \vdash e : \tau_1\ [\tau_3']\ \tau_4'}{\Gamma \vdash fe : \tau_2\ [\tau_1']\ \tau_2'}\ \text{APP}$$

$$\frac{\Gamma, f : (\tau\ {}_{\tau'} \to {}_{\tau'}\tau_1); \tau_3 \vdash e : \tau_3; \tau_2}{\Gamma; \tau_1 \vdash \mathcal{S}f.e : \tau; \tau_2}\ \text{SHIFT} \qquad \frac{\dfrac{\dfrac{\Gamma, f : \tau \xrightarrow{[\tau']\ \tau'} \tau_1 \vdash e : \tau_3\ [\tau_3]\ \tau_2}{\Gamma, f : \tau \xrightarrow{[\tau']\ \tau'} \tau_1 \vdash \langle e \rangle : \tau_2}\ \text{RESET0}}{\Gamma \vdash \mathcal{S}_0 f.\langle e \rangle : \tau\ [\tau_1\ [\tau']\ \tau']\ \tau_2}\ \text{SHIFT0} \quad \dfrac{\dots}{\tau\ [\tau_1\ [\tau']\ \tau']\ \tau_2 \le \tau\ [\tau_1]\ \tau_2}}{\Gamma \vdash \mathcal{S}_0 f.\langle e \rangle : \tau\ [\tau_1]\ \tau_2}\ \text{SUB}$$

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

Figure 11. Embedding of Danvy and Filinski's type system $\lambda^{\mathcal{S}}_{\to}$ into $\lambda^{\mathcal{S}_0}_{\le}$

to the standard CPS translations [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 $\lambda^{\mathcal{S}_0}_{\le}$, obtained using the translation inductively defined in Figure 11.

Lemma 14. *Let D_e be a typing derivation for the term e in $\lambda^{\mathcal{S}}_{\to}$. Then the following equalities hold:*

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

As a corollary, we can prove the following theorem by straightforward induction:

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

The untyped CPS translations of Section 2.2 also induce a CPS translation for *shift* and *reset*. For instance, the curried CPS translation of Figure 2 yields:

$$\begin{array}{rcl} \llbracket \mathcal{S}f.e \rrbracket & = & \lambda f.\llbracket e \rrbracket\,(\lambda x.\lambda k.k\,x) \\ \llbracket \langle e \rangle \rrbracket & = & \llbracket e \rrbracket(\lambda x.\lambda k.k\,x) \end{array}$$

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

$$\begin{array}{rcl} \langle v \rangle & \rightsquigarrow & v \\ \langle K[\mathcal{S}f.e] \rangle & \rightsquigarrow & \langle e[f/\lambda x.\langle K[x] \rangle] \rangle \end{array}$$

it generates a different equational theory than the one of the standard CPS translation for *shift* and *reset* [22]. For example, the equation

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

is not sound with respect to the presented CPS translation. So in a sense, by using the CPS translation for *shift*$_0$ and *reset*$_0$ we have obtained a variant of *shift* and *reset*. However, when considered in the type system with subtyping, *shift*$_0$ and *reset*$_0$ give rise to the standard *shift* and *reset*, as demonstrated by Lemma 14.

Each of the CPS translations of Section 2.2 determines a definition of *shift*$_0$ and *reset*$_0$ in the corresponding continuation monad. Since we can represent arbitrary computational monads in direct style using *shift* and *reset* [18], we obtain two static simulations of *shift*$_0$ and *reset*$_0$. The simulation obtained from the uncurried translation of Figure 3 is similar to the one presented by Shan [32], but the one obtained from the translation of Figure 2 is novel:

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

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):

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

Next, we can write a translation on typing derivations, which satisfies the following theorem:

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 $\lambda^{\mathcal{S}_0}_{\le}$.*

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

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

is a valid term and cannot be typed in neither of the two systems. However, it translates without changes to $\lambda^{\mathcal{S}_0}_{\le}$, where it is well

$$\frac{}{\epsilon \le \epsilon} \qquad \frac{\tau \le \tau'}{\epsilon \le [\tau]\,\tau'} \qquad \frac{\tau_2 \le \tau_1 \qquad \tau_1' \le \tau_2'}{[\tau_1]\,\tau_1' \le [\tau_2]\,\tau_2'}$$

$$\frac{\tau \le \tau' \qquad \sigma \le \sigma'}{\tau\,\sigma \le \tau'\,\sigma'} \qquad \frac{}{\alpha \le \alpha} \qquad \frac{\tau_2' \le \tau_1' \qquad \tau_1\,\sigma_1 \le \tau_2\,\sigma_2}{\tau_1' \xrightarrow{\sigma_1} \tau_1 \le \tau_2' \xrightarrow{\sigma_2} \tau_2}$$

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

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

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

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

Figure 12. Type system for $\lambda^{\mathcal{S}_0}$ with flat effect annotations ($\lambda^{\mathcal{S}}_{\le}$)

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.

An interesting feature of both translations 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 $\lambda^{\mathcal{S}_0}_{\le}$ type system, which enforces this restriction syntactically:

$$\begin{array}{lcl} \tau & ::= & \alpha \mid \tau \xrightarrow{\sigma} \tau \\ \sigma & ::= & \epsilon \mid [\tau]\,\tau \end{array}$$

The typing rules for this system (called $\lambda^{\mathcal{S}}_{\le}$) are shown in Figure 12.

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 semantics of *shift*. We can thus view this type system as an exotic type system for the *shift* operator (that is why we call it $\lambda^{\mathcal{S}}_{\le}$).

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 [31] for the programming language Scala. The presentation in [31] 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 $\lambda^{\mathcal{S}}_{\le}$) 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 [31] 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. This reinforces our opinion that Scala implements the restricted *shift*$_0$/*reset*$_0$ operators, which only happen to coincide with *shift*/*reset*.

6.2 Substructural type system of Kiselyov and Shan

The type system for *shift*$_0$/*reset*$_0$ introduced by Kiselyov and Shan [26] 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 a clever CPS translation, in Kiselyov and Shan's work it is just an entailment of the type system.

Our type system can be embedded in the type system of Kiselyov and Shan. A converse translation appears to be impossible because their language is richer: it includes an operator \$, which is a generalization of *reset*$_0$ and, in a sense, the inverse of the *shift*$_0$ operator. Because our language does not differentiate between functions and contexts, it is currently not clear how to include this operator in it, so the subject requires further study.

7. Future work

The type system presented here is monomorphic. We are planning to explore how adding polymorphism (*let*-polymorphism, system F-like explicit polymorphism) affects the system. 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, we will need to store constraints about generalized variables in the polymorphic types, as in [21].

Because of the similarities of the system presented here to the Scala implementation of delimited continuations [31], we are going to investigate whether it would be possible to implement our system in Scala. 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]]`.

Another important aspect of this work that has not been researched yet are equational theories of $shift_0$ and $reset_0$ in the untyped and typed settings as well as of the variant of *shift* and *reset* in the untyped setting, described in Section 5. In particular, it seems vital to understand the role of the subtyping relation in the equational theories of the typed control operators.

8. 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* [31]). As shown in Section 4, the full power of $shift_0$ and $reset_0$ can be very useful and we believe that there are other interesting applications of these operators waiting to be discovered.

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.

Acknowledgments

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 under control. We also thank the anonymous referees and the PC members of ICFP'11 for their valuable comments on the presentation and contents of this article. Hugo Herbelin made the observation about the difference between the standard equational theory for *shift* and *reset* and the one induced by the untyped CPS translation of Section 2.2.

This work has been partially supported by the MNiSW grant number N N206 357436, 2009-2011.

References

[1] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In D. Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19, Uppsala, Sweden, Aug. 2003. ACM Press.

[2] K. Asai and Y. Kameyama. Polymorphic delimited continuations. In *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, Dec. 2007.

[3] V. Balat and O. Danvy. Memoization in type-directed partial evaluation. In D. Batory, C. Consel, and W. 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, Oct. 2002. Springer-Verlag.

[4] C. Barker. Continuations in natural language. In H. 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, Jan. 2004.

[5] M. Biernacka and D. 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, Apr. 2009.

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

[7] M. Biernacka and O. Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theoretical Computer Science*, 375(1-3):76–108, 2007.

[8] M. Biernacka and O. Danvy. A concrete framework for environment machines. *ACM Transactions on Computational Logic*, 9(1):1–30, 2007.

[9] M. Biernacka, D. Biernacki, and O. Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science*, 1(2:5):1–39, Nov. 2005. A preliminary version was presented at the Fourth ACM SIGPLAN Workshop on Continuations (CW'04).

[10] D. Biernacki, O. Danvy, and K. 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] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In R. Sethi, editor, *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, Jan. 1977. ACM Press.

[12] O. Danvy. Type-directed partial evaluation. In G. L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, Jan. 1996. ACM Press.

[13] O. Danvy and A. Filinski. A functional abstraction of typed contexts. DIKU Rapport 89/12, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, July 1989.

[14] O. Danvy and A. Filinski. Abstracting control. In M. Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.

[15] P. Dybjer and A. Filinski. Normalization and partial evaluation. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *Applied Semantics – Advanced Lectures*, number 2395 in Lecture Notes in Computer Science, pages 137–192, Caminha, Portugal, Sept. 2000. Springer-Verlag.

[16] R. K. Dybvig, S. Peyton-Jones, and A. Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 17 (6):687–730, 2007.

[17] M. Felleisen. The theory and practice of first-class prompts. In J. Ferrante and P. Mager, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190, San Diego, California, Jan. 1988. ACM Press.

[18] A. Filinski. Representing monads. In H.-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 446–457, Portland, Oregon, Jan. 1994. ACM Press.

[19] A. Filinski. Representing layered monads. In A. Aiken, editor, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 175–188, San Antonio, Texas, Jan. 1999. ACM Press.

[20] R. Harper, B. F. Duba, and D. MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, Oct. 1993. A preliminary version was presented at the Eighteenth Annual ACM Symposium on Principles of Programming Languages (POPL 1991).

[21] F. Henglein. Syntactic properties of polymorphic subtyping. Technical Report Semantics Report D-293, DIKU, Computer Science Department, University of Copenhagen, May 1996.

[22] Y. Kameyama and M. Hasegawa. A sound and complete axiomatization of delimited continuations. In O. 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, Aug. 2003. ACM Press.

[23] R. Kelsey, W. Clinger, and J. Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[24] J. Kim, K. Yi, and O. Danvy. Assessing the overhead of ML exceptions by selective CPS transformation. In G. Morrisett, editor, *Record of the 1998 ACM SIGPLAN Workshop on ML and its Applications*, Baltimore, Maryland, Sept. 1998.

[25] O. Kiselyov and C. Shan. Delimited continuations in operating systems. In B. Kokinov, D. C. Richardson, T. R. Roth-Berghofer, and L. 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, Aug. 2007. Springer.

[26] O. Kiselyov and C. Shan. A substructural type system for delimited continuations. In S. R. D. 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.

[27] L. R. Nielsen. A selective CPS transformation. In S. Brookes and M. 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.

[28] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[29] J. 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 [30].

[30] J. C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.

[31] T. Rompf, I. Maier, and M. Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. In A. Tolmach, editor, *Proceedings of the 2009 ACM SIGPLAN International Conference on Functional Programming (ICFP'09)*, pages 317–328, Edinburgh, UK, Aug. 2009. ACM Press.

[32] C. Shan. A static simulation of dynamic delimited control. *Higher-Order and Symbolic Computation*, 20(4):371–401, 2007.

[33] E. Sumii. An implementation of transparent migration on standard Scheme. In M. Felleisen, editor, *Proceedings of the Workshop on Scheme and Functional Programming*, Technical Report 00-368, Rice University, pages 61–64, Montréal, Canada, Sept. 2000.