Selective CPS Transformation for Shift and Reset

Kenichi Asai
Ochanomizu University
Tokyo, Japan
asai@is.ocha.ac.jp

Chihiro Uehara
Ochanomizu University
Tokyo, Japan
uehara.chihiro@is.ocha.ac.jp

Abstract

This paper presents a selective CPS transformation for a program that uses control operators, shift and reset, introduced by Danvy and Filinski. By selectively CPS-transforming a program, we can execute a program with shift and reset in a standard functional language without support for control operators. We introduce a constraint-based type inference system that annotates the parts that are captured by shift and thus require CPS transformation. We show that the best annotation does not exist in general, and present a constraint solving algorithm that is reasonably efficient. The selective CPS transformation is defined over annotated terms and its correctness is proven. Finally, experimental results show that selective CPS transformation does improve performance compared to the standard CPS transformation.

CCS Concepts • **Theory of computation → Control primitives**;

Keywords Selective CPS transformation, delimited continuations, type system, functional languages

ACM Reference Format:
Kenichi Asai and Chihiro Uehara. 2018. Selective CPS Transformation for Shift and Reset. In *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'18).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3162069

1 Introduction

Control operators, shift and reset, as proposed by Danvy and Filinski [7] enable us to manipulate control of a program without transforming the program into continuation-passing style (CPS). They have been extensively studied, including type system [6], its extension to let-polymorphism [2], axiomatization [15], and CPS hierarchy [4] to name a few. They are also useful in many applications, such as non-deterministic programming [7], representing monads [13], let-insertion in partial evaluation [19, 28], and typed printf [1, 5].

However, it is not easy to implement shift and reset in the standard functional languages, especially in a typed setting. Since the presence of shift and reset imposes restriction on the type of the delimited context, a type system for shift and reset needs to take the type of the delimited context (or *the answer type*) into account. In fact, OchaCaml[1] [21], a direct implementation of shift/reset on top of Caml Light, replaces the type system of Caml Light completely with a new one that incorporates answer types. Such a replacement becomes less and less realistic for full-fledged typed functional languages like OCaml. As such, Kiselyov [17] implements the Delimcc Library that implements control operators including shift and reset as a library module independent of the host OCaml system to ease maintenance. However, since he does not modify the type system of OCaml, the so-called answer type modification [2, 6] is not allowed, restricting the applicability. One needs an additional manual trick [18] to write a program with answer type modification.

In this paper, we take a different approach. We transform a source program with shift and reset into a program without via CPS transformation. However, we do not want to do it all the time. When shift and reset are not used, we want to keep the program in direct style for faster execution. To this end, we transform the source program into CPS selectively: we transform the parts that are captured by shift and leave the other parts in direct style. Thus, a program that does not use shift/reset is executed exactly the same as before, while a program that uses shift/reset is selectively transformed into CPS so that it can be executed in the standard functional language without support for shift/reset. The selective CPS transformation allows full answer type modification: the source program is typed according to the type system with answer types while the target program is typed according to the standard type system.

To perform the selective CPS transformation, we need to identify the parts that require CPS transformation. We introduce a constraint-based type inference system similar in spirit to constraint-based binding-time analysis [14]. Interestingly, we will show that there is no best annotation in general with a concrete example. The contributions of the paper are summarized as follows:

- We present a type system that assigns purity annotations to indicate if CPS transformation is required. The

PEPM'18, January 8–9, 2018, Los Angeles, CA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5587-2/18/01…$15.00
https://doi.org/10.1145/3162069

[1] `http://pllab.is.ocha.ac.jp/~asai/OchaCaml/`

typability of our system is shown to be equivalent to Asai and Kameyama's system [2] but our type system provides us with a finer classification of purity.
- We show that the best annotation does not exist in general. Accordingly, we present a reasonably efficient type inference system.
- We show a selective CPS transformation for annotated terms and prove its correctness, i.e., the selective CPS transformation preserves the meaning of a term.
- We show some experimental results to show that selective CPS transformation does provide us with faster execution than the standard CPS transformation.

Rompf, Maier, and Odersky [26] implement shift/reset in Scala by a type-directed selective CPS transformation. Our work is a reformulation of their work in the functional setting. By introducing constraint-based type inference, we avoid user annotations required in Scala. The effect of selective CPS transformation can be big. Leijen [20] reports that by employing selective CPS transformation (to compile away algebraic effects in the language Koka), only less than 20% of the core library functions are required to be in CPS.

In the next section, we define types, terms, and typing rules with purity annotations that we use throughout the paper. Due to the space limitation, we did not include an introduction to shift and reset. We refer the reader to a tutorial [3]. In Section 3, we investigate the relationship between our system and Asai and Kameyama's system. Section 4 presents a type inference algorithm with constraint solving on purity annotations. Selective CPS transformation is shown in Section 5, whose correctness is proved in Section 6. Experimental results are in Section 7. Section 8 mentions a use of library functions. We discuss related work in Section 9 and conclude in Section 10.

2 Annotated Types, Terms, Typing Rules

Types and terms are shown in Figure 1. We follow Asai and Kameyama's system [2] and extend it with purity annotations.

An annotation is either p (pure) indicating that shift is not used and thus CPS transformation is not required, or i (impure) indicating that shift might have been used and thus CPS transformation is required. We assume inequality $\mathsf{p} < \mathsf{i}$ between the two annotations, meaning that a pure term can always be regarded as impure (by unnecessarily transforming the pure term into CPS).

A type is either a type variable α, a base type b (such as boolean and integer), or $\tau_2 \to \tau_1@\mathrm{cps}[\tau_3, \tau_4, a]$.[2] The latter is basically a function type from τ_2 to τ_1, but application of this function causes the answer type to change from τ_3 to τ_4. The purity annotation a in the type indicates if shift is used during the application of this function. If a is p, the two answer types, τ_3 and τ_4, must always be the same. Later, we will enforce this constraint in the typing rules.

$$\begin{array}{lcll}
a & ::= & \mathsf{p} \mid \mathsf{i} & \text{purity annotation} \\
\tau & ::= & \alpha \mid b \mid \tau_2 \to \tau_1@\mathrm{cps}[\tau_3, \tau_4, a] & \text{monomorphic type} \\
\sigma & ::= & \tau \mid \forall \alpha . \sigma & \text{polymorphic type} \\
\Gamma & ::= & \cdot \mid \Gamma, x : \sigma & \text{type environment} \\
v & ::= & c \mid x \mid \lambda^a x.\, A_1 \mid \mathsf{fix}^a\, f.x.\, A_1 & \text{value} \\
e & ::= & v_1 \mid A_1 \mathbin{@^a} A_2 \mid \mathcal{S}^a k.\, A_1 \mid \langle A_1 \rangle \mid & \text{term} \\
 & & \mathsf{let}\; x = {v_1}^{\mathsf{p}}\; \mathsf{in}\; A_2 \mid \mathsf{if}\; A_1\; \mathsf{then}\; A_2\; \mathsf{else}\; A_3 & \\
A & ::= & e^a & \text{annotated term}
\end{array}$$

Figure 1. Types and terms

A term consists of the standard λ-calculus terms extended with delimited control operators, shift and reset, together with constants, fixed points, polymorphic let expressions (with value restriction), and conditionals. Every subterm is annotated with p or i. In addition, $\lambda^a x.\, A_1$, $\mathsf{fix}^a\, f.x.\, A_1$, $A_1 \mathbin{@^a} A_2$, and $\mathcal{S}^a k.\, A_1$ carry additional annotation a. We will explain its role with typing rules below.

Definition 2.1. A term e is *syntactically pure* if e is either a value v or of the form $\langle A_1 \rangle$ for some A_1.

Typing rules are shown in Figure 2. Judgement has the form $\Gamma \vdash e^a : \tau_1\, @\mathrm{cps}[\tau_2, \tau_3, a]$[3] which reads: under the type environment Γ, the annotated term e^a has type τ_1 whose execution changes the answer type from τ_2 to τ_3. Since the two occurrences of a in the judgement are always the same, the purity annotation in a term is redundant: we can always recover it from the typing derivation. However, we will keep the annotation in a term, because selective CPS transformation is defined over annotated terms. Equivalently, we could define selective CPS transformation over typing derivations.

The typing rules are a natural extension of those of Asai and Kameyama [2] to express purity information. We explain the extended parts below. Precise relationship to Asai and Kameyama's typing rules is discussed in Section 3. Syntactically pure terms are given the annotation p, and the two answer types are required to be the same. As for $(\lambda^{a_2} x.\, {e_1}^{a_1})^{\mathsf{p}}$, although the abstraction itself is pure, its body e_1 can be impure. When it is impure, a_2 also becomes i from $a_1 \le a_2$, and the type of the abstraction becomes $(\tau_2 \to \tau_1@\mathrm{cps}[\tau_3, \tau_4, a_2])$. Here, the two answer types, τ_3 and τ_4, can be different. When e_1 is pure, on the other hand, τ_3 and τ_4 must be the same. This constraint is expressed by $\tau_3 \ne \tau_4 \Rightarrow a_1 = \mathsf{i}$. Even if the body e_1 is pure, we sometimes want to treat it as impure, as in $\lambda x.\, x$ in the following term:

$$\mathsf{if\; true\; then}\; \lambda x.\, x\; \mathsf{else}\; \lambda x.\, \mathcal{S}k.\, x$$

[2] We borrowed the notation from Rompf, Maier, and Odersky [26] here. If we ignore the purity annotation, it corresponds to $\tau_2/\tau_3 \to \tau_1/\tau_4$ in Danvy and Filinski's notation [6].

[3] If we ignore the purity annotation, it corresponds to $\Gamma, \tau_2 \vdash e : \tau_1, \tau_3$ in Danvy and Filinski's notation [6].

$$\boxed{\Gamma \vdash e^a : \tau_1 \mathbin{@}\mathrm{cps}[\tau_2, \tau_3, a]} \qquad \frac{(x : \sigma \in \Gamma \text{ and } \sigma > \tau_1)}{\Gamma \vdash x^{\mathrm{p}} : \tau_1 \mathbin{@}\mathrm{cps}[\tau_2, \tau_2, \mathrm{p}]}\ (\textsc{Var}) \qquad \frac{(c \text{ is a constant of basic type } b)}{\Gamma \vdash c^{\mathrm{p}} : b \mathbin{@}\mathrm{cps}[\tau_1, \tau_1, \mathrm{p}]}\ (\textsc{Const})$$

$$\frac{\Gamma, x : \tau_2 \vdash {e_1}^{a_1} : \tau_1 \mathbin{@}\mathrm{cps}[\tau_3, \tau_4, a_1] \quad a_1 \le a_2 \quad \tau_3 \ne \tau_4 \Rightarrow a_1 = \mathrm{i}}{\Gamma \vdash (\lambda^{a_2} x.\, {e_1}^{a_1})^{\mathrm{p}} : (\tau_2 \to \tau_1 @\mathrm{cps}[\tau_3, \tau_4, a_2]) \mathbin{@}\mathrm{cps}[\tau_5, \tau_5, \mathrm{p}]}\ (\textsc{Fun})$$

$$\frac{\Gamma, f : (\tau_2 \to \tau_1 @\mathrm{cps}[\tau_3, \tau_4, a_1]), x : \tau_2 \vdash {e_1}^{a_1} : \tau_1 \mathbin{@}\mathrm{cps}[\tau_3, \tau_4, a_1] \quad a_1 \le a_2 \quad \tau_3 \ne \tau_4 \Rightarrow a_1 = \mathrm{i}}{\Gamma \vdash (\mathrm{fix}^{a_2} f.x.\, {e_1}^{a_1})^{\mathrm{p}} : (\tau_2 \to \tau_1 @\mathrm{cps}[\tau_3, \tau_4, a_2]) \mathbin{@}\mathrm{cps}[\tau_5, \tau_5, \mathrm{p}]}\ (\textsc{Fix})$$

$$\frac{\begin{array}{c} a_1 \le a \quad a_2 \le a \quad a_3 \le a \quad \tau_5 \ne \tau_6 \Rightarrow a_1 = \mathrm{i} \quad \tau_4 \ne \tau_5 \Rightarrow a_2 = \mathrm{i} \quad \tau_3 \ne \tau_4 \Rightarrow a_3 = \mathrm{i} \\ \Gamma \vdash {e_1}^{a_1} : (\tau_2 \to \tau_1 @\mathrm{cps}[\tau_3, \tau_4, a_3]) \mathbin{@}\mathrm{cps}[\tau_5, \tau_6, a_1] \quad \Gamma \vdash {e_2}^{a_2} : \tau_2 \mathbin{@}\mathrm{cps}[\tau_4, \tau_5, a_2] \end{array}}{\Gamma \vdash ({e_1}^{a_1} \mathbin{@}^{a_3} {e_2}^{a_2})^a : \tau_1 \mathbin{@}\mathrm{cps}[\tau_3, \tau_6, a]}\ (\textsc{App})$$

$$\frac{\Gamma, k : \forall\alpha.(\tau_3 \to \tau_4 @\mathrm{cps}[\alpha, \alpha, a_2]) \vdash {e_1}^{a_1} : \tau_1 \mathbin{@}\mathrm{cps}[\tau_1, \tau_2, a_1] \quad \tau_1 \ne \tau_2 \Rightarrow a_1 = \mathrm{i}}{\Gamma \vdash (S^{a_2} k.\, {e_1}^{a_1})^{\mathrm{i}} : \tau_3 \mathbin{@}\mathrm{cps}[\tau_4, \tau_2, \mathrm{i}]}\ (\textsc{Shift})$$

$$\frac{\Gamma \vdash {e_1}^{a_1} : \tau_1 \mathbin{@}\mathrm{cps}[\tau_1, \tau_2, a_1] \quad \tau_1 \ne \tau_2 \Rightarrow a_1 = \mathrm{i}}{\Gamma \vdash \langle {e_1}^{a_1} \rangle^{\mathrm{p}} : \tau_2 \mathbin{@}\mathrm{cps}[\tau_3, \tau_3, \mathrm{p}]}\ (\textsc{Reset})$$

$$\frac{\Gamma \vdash {v_1}^{\mathrm{p}} : \tau_1 \mathbin{@}\mathrm{cps}[\tau_5, \tau_5, \mathrm{p}] \quad \Gamma, x : \mathrm{Gen}(\tau_1; \Gamma) \vdash {e_2}^{a_2} : \tau_2 \mathbin{@}\mathrm{cps}[\tau_3, \tau_4, a_2] \quad a_2 \le a \quad \tau_3 \ne \tau_4 \Rightarrow a_2 = \mathrm{i}}{\Gamma \vdash (\mathrm{let}\, x = {v_1}^{\mathrm{p}} \,\mathrm{in}\, {e_2}^{a_2})^a : \tau_2 \mathbin{@}\mathrm{cps}[\tau_3, \tau_4, a]}\ (\textsc{Let})$$

$$\frac{\begin{array}{c} a_1 \le a \quad a_2 \le a \quad a_3 \le a \quad \tau_3 \ne \tau_4 \Rightarrow a_1 = \mathrm{i} \quad \tau_2 \ne \tau_3 \Rightarrow a_2 = \mathrm{i} \quad \tau_2 \ne \tau_3 \Rightarrow a_3 = \mathrm{i} \\ \Gamma \vdash {e_1}^{a_1} : \mathrm{bool} \mathbin{@}\mathrm{cps}[\tau_3, \tau_4, a_1] \quad \Gamma \vdash {e_2}^{a_2} : \tau_1 \mathbin{@}\mathrm{cps}[\tau_2, \tau_3, a_2] \quad \Gamma \vdash {e_3}^{a_3} : \tau_1 \mathbin{@}\mathrm{cps}[\tau_2, \tau_3, a_3] \end{array}}{\Gamma \vdash (\mathrm{if}\, {e_1}^{a_1} \,\mathrm{then}\, {e_2}^{a_2} \,\mathrm{else}\, {e_3}^{a_3})^a : \tau_1 \mathbin{@}\mathrm{cps}[\tau_2, \tau_4, a]}\ (\textsc{If})$$

Figure 2. Typing rules

Such cases are accounted for by the inequality $a_1 < a_2$.[4] Even if the body of abstraction is pure, it can be treated as impure externally. The situation is similar in (Fix). The annotation a_1 represents the purity of the recursive function f in its body e_1, while a_2 represents how the recursive function is handled externally.

Typing rules for applications, conditionals, and let expressions simply propagate annotations of the subterms to the annotation of the whole term. If any of the subterms are impure, so is the whole term. The inequality constraints allow subterms to be pure, even if the whole term is impure. This is how a pure term can be embedded into an impure context.

Finally, (Shift) is the only rule that enforces the impure annotation i. Without this rule, we could satisfy all the constraints by assigning p to all the annotations and equating all the adjacent answer types. In (Shift), the variable k has the annotation a_2 rather than p, although the captured continuation is always pure. This is because we sometimes want to treat k as impure, just like $\lambda x.\, x$ in the above example.

[4] The inequality in (If) does *not* account for this case. It can change only the topmost annotation. Here, we need to change the annotation of the body of λ-abstraction.

We can show various simple properties of the typing rules. First, syntactically pure terms are answer-type polymorphic, i.e., we can freely replace the two answer types.

Proposition 2.2. *For any syntactically pure term e, if $\Gamma \vdash e^a : \tau_1 \mathbin{@}\mathrm{cps}[\tau_2, \tau_3, a]$, then $a = \mathrm{p}$, $\tau_2 = \tau_3$, and $\Gamma \vdash e^{\mathrm{p}} : \tau_1 \mathbin{@}\mathrm{cps}[\tau_2', \tau_2', \mathrm{p}]$ for any type τ_2'.*

Proof. By simple inspection of the typing rules for syntactically pure terms. □

We cannot extend this property to an arbitrary term e, because we do not have polymorphism for λ-bound variables. For example, consider:

$$(\lambda f.\, \mathrm{let}\, v = \langle f \mathbin{@} 0 \rangle + 1 \,\mathrm{in}\, f) \mathbin{@} (\lambda x.\, x)$$

Although this term is pure (as shift is not used), the answer type of the result f is fixed to integer, because of the first use of f. To extend the above property to an arbitrary term, as done by Thielecke [27], we need first-class polymorphism, such as in System F.

The next proposition states that the two answer types are equal for pure terms. This proposition ensures that the typing rules are equipped with enough constraints of the form $\tau_1 \ne \tau_2 \Rightarrow a = \mathrm{i}$.

Proposition 2.3. *If* $\Gamma \vdash e^a : \tau_1 \,@\mathrm{cps}[\tau_2, \tau_3, a]$*, then* $\tau_2 \neq \tau_3 \Rightarrow a = \mathrm{i}$*. In particular, if* $\Gamma \vdash e^{\mathrm{p}} : \tau_1 \,@\mathrm{cps}[\tau_2, \tau_3, \mathrm{p}]$*, then* $\tau_2 = \tau_3$*.*

Proof. By simple induction on the derivation of $\Gamma \vdash e^a : \tau_1 \,@\mathrm{cps}[\tau_2, \tau_3, a]$. □

Finally, non-syntactically pure terms can be made impure. This proposition ensures that we have inserted enough inequality constraints in the typing rules, so that we can always coerce pure terms into impure ones, if the term is not syntactically pure.

Proposition 2.4. *For any term e that is not syntactically pure, if* $\Gamma \vdash e^{\mathrm{p}} : \tau_1 \,@\mathrm{cps}[\tau_2, \tau_2, \mathrm{p}]$*, then* $\Gamma \vdash e^{\mathrm{i}} : \tau_1 \,@\mathrm{cps}[\tau_2, \tau_2, \mathrm{i}]$*.*

Proof. By simple induction on the typing rules for terms that are not syntactically pure. □

3 Relationship to Asai and Kameyama's System

In this section, we show that the typing rules in Figure 2 are a conservative extension of Asai and Kameyama's typing rules [2]. To be more precise, a term is typable in our system if and only if it is typable in Asai and Kameyama's system, restricted to value restriction for let expressions instead of purity restriction.[5] This does not mean that our system is useless. Asai and Kameyama's system classifies only syntactically pure terms as pure. Our system refines the notion of purity so that it can classify more terms as pure.

Figure 3 defines types and terms for Asai and Kameyama's system. Types and terms are obtained by simply removing all the purity annotations from our system, as formalized in Figure 4. Typing rules for Asai and Kameyama's system are in Figure 5. Judgements are split into two kinds: pure ones and impure ones. The pure judgement, $\Gamma \vdash_p^{AK} e : \tau_1$, is for a syntactically pure term e and reads: under type environment Γ, the syntactically pure term e has type τ_1. The impure judgement, $\Gamma \vdash^{AK} e : \tau_1 \,@\mathrm{cps}[\tau_2, \tau_3]$, is for an arbitrary term e and reads: under type environment Γ, the term e has type τ_1 whose execution changes the answer type from τ_2 to τ_3. Typing rules are obtained from Figure 2 by removing purity annotations and by removing the two answer types for syntactically pure terms. To coerce a pure judgement into an impure one, (ExpAK) is introduced.

Since pure judgements are used only for syntactically pure terms in Asai and Kameyama's system, and since pure judgements can be turned into impure ones via (ExpAK), we can always turn our typing derivation into Asai and Kameyama's system simply by forgetting all the purity annotations.

Theorem 3.1. (1) *For any syntactically pure term e, if* $\Gamma \vdash e^{\mathrm{p}} : \tau_1 \,@\mathrm{cps}[\tau_2, \tau_2, \mathrm{p}]$*, then* $|\Gamma| \vdash_p^{AK} |e^{\mathrm{p}}| : |\tau_1|$ *and hence* $|\Gamma| \vdash^{AK} |e^{\mathrm{p}}| : |\tau_1| \,@\mathrm{cps}[|\tau_2|, |\tau_2|]$ *using* (ExpAK). (2) *For any term e that is not syntactically pure, if* $\Gamma \vdash e^a : \tau_1 \,@\mathrm{cps}[\tau_2, \tau_3, a]$*, then* $|\Gamma| \vdash^{AK} |e^a| : |\tau_1| \,@\mathrm{cps}[|\tau_2|, |\tau_3|]$*.*

Proof. By simple induction on the derivation of $\Gamma \vdash e^a : \tau_1 \,@\mathrm{cps}[\tau_2, \tau_3, a]$. □

$$
\begin{array}{rcll}
\tau & ::= & \alpha \mid b \mid \tau_2 \to \tau_1 @\mathrm{cps}[\tau_3, \tau_4] & \text{monomorphic type} \\
\sigma & ::= & \tau \mid \forall \alpha.\sigma & \text{polymorphic type} \\
\Gamma & ::= & \cdot \mid \Gamma, x : \sigma & \text{type environment} \\
v & ::= & c \mid x \mid \lambda x.\, e_1 \mid \mathsf{fix}\, f.x.\, e_1 & \text{value} \\
e & ::= & v_1 \mid e_1 \,@\, e_2 \mid Sk.\, e_1 \mid \langle e_1 \rangle \mid & \text{term} \\
 & & \mathsf{let}\; x = v_1 \;\mathsf{in}\; e_2 \mid \mathsf{if}\; e_1 \;\mathsf{then}\; e_2 \;\mathsf{else}\; e_3 &
\end{array}
$$

Figure 3. Types and terms for Asai and Kameyama's system [2]

$$
\begin{array}{rcl}
|\alpha| & = & \alpha \\
|b| & = & b \\
|\tau_2 \to \tau_1 @\mathrm{cps}[\tau_3, \tau_4, a]| & = & |\tau_2| \to |\tau_1| @\mathrm{cps}[|\tau_3|, |\tau_4|] \\
|\forall \alpha.\sigma| & = & \forall \alpha.|\sigma| \\
|\cdot| & = & \cdot \\
|\Gamma, x : \sigma| & = & |\Gamma|, x : |\sigma| \\
|x^a| & = & x \\
|c^a| & = & c \\
|(\lambda^{a_2} x.\, e_1{}^{a_1})^a| & = & \lambda x.\, |e_1{}^{a_1}| \\
|(\mathsf{fix}^{a_2}\, f.x.\, e_1{}^{a_1})^a| & = & \mathsf{fix}\, f.x.\, |e_1{}^{a_1}| \\
|(e_1{}^{a_1} \,@^{a_3}\, e_2{}^{a_2})^a| & = & |e_1{}^{a_1}| \,@\, |e_2{}^{a_2}| \\
|(S^{a_2} k.\, e_1{}^{a_1})^{\mathrm{i}}| & = & Sk.\, |e_1{}^{a_1}| \\
|\langle e_1{}^{a_1} \rangle^a| & = & \langle |e_1{}^{a_1}| \rangle \\
|(\mathsf{let}\; x = v_1{}^{\mathrm{p}} \;\mathsf{in}\; e_2{}^{a_2})^a| & = & \mathsf{let}\; x = |v_1{}^{\mathrm{p}}| \;\mathsf{in}\; |e_2{}^{a_2}| \\
|(\mathsf{if}\; e_1{}^{a_1} \;\mathsf{then}\; e_2{}^{a_2} \;\mathsf{else}\; e_3{}^{a_3})^a| & = & \mathsf{if}\; |e_1{}^{a_1}| \;\mathsf{then}\; |e_2{}^{a_2}| \;\mathsf{else}\; |e_3{}^{a_3}|
\end{array}
$$

Figure 4. Annotation erasure

To show the other direction, we need to recover appropriate purity annotations. We first define the attachment of impure annotations to types and type environments as follows:

$$
\begin{array}{rcl}
\alpha^{\mathrm{i}} & = & \alpha \\
b^{\mathrm{i}} & = & b \\
(\tau_2 \to \tau_1 @\mathrm{cps}[\tau_3, \tau_4])^{\mathrm{i}} & = & \tau_2{}^{\mathrm{i}} \to \tau_1{}^{\mathrm{i}} @\mathrm{cps}[\tau_3{}^{\mathrm{i}}, \tau_4{}^{\mathrm{i}}, \mathrm{i}] \\
(\forall \alpha.\sigma)^{\mathrm{i}} & = & \forall \alpha.\sigma^{\mathrm{i}} \\
(\cdot)^{\mathrm{i}} & = & \cdot \\
(\Gamma, x : \sigma)^{\mathrm{i}} & = & \Gamma^{\mathrm{i}}, x : \sigma^{\mathrm{i}}
\end{array}
$$

We can then turn a typing derivation in Asai and Kameyama's system into ours. The recovered annotations are all impure, except for syntactically pure terms which are fixed to be pure.

Theorem 3.2. (1) *If* $\Gamma \vdash_p^{AK} e : \tau_1$*, then for any type* τ_2*, we have* $\Gamma^{\mathrm{i}} \vdash e^{\mathrm{p}} : \tau_1{}^{\mathrm{i}} \,@\mathrm{cps}[\tau_2{}^{\mathrm{i}}, \tau_2{}^{\mathrm{i}}, \mathrm{p}]$ *for some* e^{p} *such that* $e = |e^{\mathrm{p}}|$*.* (2) *If* $\Gamma \vdash^{AK} e : \tau_1 \,@\mathrm{cps}[\tau_2, \tau_2]$ *and e is syntactically*

[5] In this paper, we ignore purity restriction and use value restriction. It should be easy to adjust our system to incorporate purity restriction.

$$\boxed{\Gamma \vdash_p^{AK} e : \tau_1} \qquad \frac{(x : \sigma \in \Gamma \text{ and } \sigma \succ \tau_1)}{\Gamma \vdash_p^{AK} x : \tau_1}\ (\textsc{VarAK})$$

$$\frac{(c \text{ is a constant of basic type } b)}{\Gamma \vdash_p^{AK} c : b}\ (\textsc{ConstAK}) \qquad \frac{\Gamma, x : \tau_2 \vdash^{AK} e_1 : \tau_1\ @\mathrm{cps}[\tau_3, \tau_4]}{\Gamma \vdash_p^{AK} \lambda x.\, e_1 : (\tau_2 \to \tau_1 @\mathrm{cps}[\tau_3, \tau_4])}\ (\textsc{FunAK})$$

$$\frac{\Gamma, f : (\tau_2 \to \tau_1 @\mathrm{cps}[\tau_3, \tau_4]), x : \tau_2 \vdash^{AK} e_1 : \tau_1\ @\mathrm{cps}[\tau_3, \tau_4]}{\Gamma \vdash_p^{AK} \mathsf{fix}\, f.x.\, e_1 : (\tau_2 \to \tau_1 @\mathrm{cps}[\tau_3, \tau_4])}\ (\textsc{FixAK}) \qquad \frac{\Gamma \vdash^{AK} e_1 : \tau_1\ @\mathrm{cps}[\tau_1, \tau_2]}{\Gamma \vdash_p^{AK} \langle e_1 \rangle : \tau_2}\ (\textsc{ResetAK})$$

$$\boxed{\Gamma \vdash^{AK} e : \tau_1\ @\mathrm{cps}[\tau_2, \tau_3]} \qquad \frac{\Gamma \vdash^{AK} e_1 : (\tau_2 \to \tau_1 @\mathrm{cps}[\tau_3, \tau_4])\ @\mathrm{cps}[\tau_5, \tau_6] \quad \Gamma \vdash^{AK} e_2 : \tau_2\ @\mathrm{cps}[\tau_4, \tau_5]}{\Gamma \vdash^{AK} e_1\ @\ e_2 : \tau_1\ @\mathrm{cps}[\tau_3, \tau_6]}\ (\textsc{AppAK})$$

$$\frac{\Gamma, k : \forall \alpha.(\tau_3 \to \tau_4 @\mathrm{cps}[\alpha, \alpha]) \vdash^{AK} e_1 : \tau_1\ @\mathrm{cps}[\tau_1, \tau_2]}{\Gamma \vdash^{AK} \mathcal{S}k.\, e_1 : \tau_3\ @\mathrm{cps}[\tau_4, \tau_2]}\ (\textsc{ShiftAK})$$

$$\frac{\Gamma \vdash_p^{AK} v_1 : \tau_1 \quad \Gamma, x : \mathrm{Gen}(\tau_1; \Gamma) \vdash^{AK} e_2 : \tau_2\ @\mathrm{cps}[\tau_3, \tau_4]}{\Gamma \vdash^{AK} \mathsf{let}\, x = v_1 \,\mathsf{in}\, e_2 : \tau_2\ @\mathrm{cps}[\tau_3, \tau_4]}\ (\textsc{LetAK}) \qquad \frac{\Gamma \vdash_p^{AK} e : \tau_1}{\Gamma \vdash^{AK} e : \tau_1\ @\mathrm{cps}[\tau_2, \tau_2]}\ (\textsc{ExpAK})$$

$$\frac{\Gamma \vdash^{AK} e_1 : \mathsf{bool}\ @\mathrm{cps}[\tau_3, \tau_4] \quad \Gamma \vdash^{AK} e_2 : \tau_1\ @\mathrm{cps}[\tau_2, \tau_3] \quad \Gamma \vdash^{AK} e_3 : \tau_1\ @\mathrm{cps}[\tau_2, \tau_3]}{\Gamma \vdash^{AK} \mathsf{if}\, e_1 \,\mathsf{then}\, e_2 \,\mathsf{else}\, e_3 : \tau_1\ @\mathrm{cps}[\tau_2, \tau_4]}\ (\textsc{IfAK})$$

Figure 5. Typing rules for Asai and Kameyama's system [2]

pure, then we have $\Gamma^{\mathrm{i}} \vdash e^{\mathrm{p}} : {\tau_1}^{\mathrm{i}}\ @\mathrm{cps}[{\tau_2}^{\mathrm{i}}, {\tau_2}^{\mathrm{i}}, \mathrm{p}]$ *for some* e^{p} *such that* $e = |e^{\mathrm{p}}|$. (3) *If* $\Gamma \vdash^{AK} e : \tau_1\ @\mathrm{cps}[\tau_2, \tau_3]$ *and* e *is not syntactically pure, then we have* $\Gamma^{\mathrm{i}} \vdash e^{\mathrm{i}} : {\tau_1}^{\mathrm{i}}\ @\mathrm{cps}[{\tau_2}^{\mathrm{i}}, {\tau_3}^{\mathrm{i}}, \mathrm{i}]$ *for some* e^{i} *such that* $e = |e^{\mathrm{i}}|$.

Proof. By simultaneous induction on the derivation of $\Gamma \vdash_p^{AK} e : \tau_1$ and $\Gamma \vdash^{AK} e : \tau_1\ @\mathrm{cps}[\tau_2, \tau_3]$. □

The above two theorems clarify the relationship between our system and Asai and Kameyama's system. Although the set of well-typed terms are equal in both the systems, our system can classify more terms as pure. In Asai and Kameyama's system, terms are classified as pure only when they are syntactically pure. In particular, applications are always impure. In our system, applications can be pure if the function part, argument part, and the body of the function part are all pure.

If we choose the most impure annotations, our system coincides with Asai and Kameyama's. What we want is an annotation that is pure as much as possible.

4 Finding Better Annotations

Given a term without annotations, we try to attach purity annotations to all the subterms that are as pure as possible. Ideally, we want to find the best (or principal) annotation, i.e., the one that is smaller than any other possible annotations. However, it turns out that such an annotation does not exist in general. In this section, we show how to find annotations that are reasonably good without introducing much computational costs. When better annotations could have been found with additional efforts, we will mention it explicitly along the development.

Given a term e, we first attach new annotation variables to all the subterms of e. We then perform standard type inference, collecting all the constraints on purity annotations along the way. We then solve the collected constraints. Constraints solving consists of four steps:

1. Transform constraints of the form $\tau_1 \neq \tau_2 \Rightarrow a = \mathrm{i}$ into ones that do not mention types.
2. Solve trivial constraints.
3. Solve remaining constraints.
4. Assign p to all the remaining annotations.

Among the four steps, the first and third ones could throw away better annotations. We describe each step in detail below.

4.1 Removing Dependence on Types

We first transform the constraints of the form $\tau_1 \neq \tau_2 \Rightarrow a = \mathrm{i}$ into either $a = \mathrm{i}$ or $a_1 \neq a_2 \Rightarrow a = \mathrm{i}$ by case analysis on τ_1 and τ_2, where a_1 and a_2 are purity annotations (not types). If τ_1 and τ_2 are the same type (e.g., bool $\neq$ bool $\Rightarrow a = \mathrm{i}$), the constraint is trivially satisfied and is removed. If τ_1 and τ_2 differ, $\tau_1 \neq \tau_2 \Rightarrow a = \mathrm{i}$ is replaced by $a = \mathrm{i}$. For example, bool $\neq$ int $\Rightarrow a = \mathrm{i}$ becomes $a = \mathrm{i}$. For a function type:

$$\tau_{12} \to \tau_{11} @\mathrm{cps}[\tau_{13}, \tau_{14}, a_1] \neq \tau_{22} \to \tau_{21} @\mathrm{cps}[\tau_{23}, \tau_{24}, a_2]$$
$$\Rightarrow a = \mathrm{i}$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash g^{\mathsf{p}} : (\mathsf{bool} \to \alpha_1 @\mathsf{cps}[\alpha_3, \alpha_4, a_3]) @\mathsf{cps}[\alpha_4, \alpha_4, \mathsf{p}]}\ (\textsc{Var}) \quad \dfrac{}{\Gamma' \vdash \mathsf{true}^{\mathsf{p}} : \mathsf{bool} @\mathsf{cps}[\alpha_4, \alpha_4, \mathsf{p}]}\ (\textsc{Const})}{\Gamma' \vdash (g^{\mathsf{p}} @^{a_3} \mathsf{true}^{\mathsf{p}})^{a_1} : \alpha_1 @\mathsf{cps}[\alpha_3, \alpha_4, a_1]}\ (\textsc{App})}{\Gamma \vdash (\lambda^{a_2} g.\, (g^{\mathsf{p}} @^{a_3} \mathsf{true}^{\mathsf{p}})^{a_1})^{\mathsf{p}} : ((\mathsf{bool} \to \alpha_1 @\mathsf{cps}[\alpha_3, \alpha_4, a_3]) \to \alpha_1 @\mathsf{cps}[\alpha_3, \alpha_4, a_2]) @\mathsf{cps}[\tau_5, \tau_5, \mathsf{p}]}\ (\textsc{Fun})$$

Collected constraints: $a_3 \le a_1$, $a_1 \le a_2$, $\alpha_3 \ne \alpha_4 \Rightarrow a_1 = \mathsf{i}$, $\alpha_3 \ne \alpha_4 \Rightarrow a_3 = \mathsf{i}$.

Figure 6. Typing derivation for $\lambda g.\, g @ \mathsf{true}$, where $\Gamma' = \Gamma, g : (\mathsf{bool} \to \alpha_1 @\mathsf{cps}[\alpha_3, \alpha_4, a_3])$

we compare recursively. If any of the components differ, the constraint becomes $a = \mathsf{i}$. If all the components are the same including purity annotations, the constraint is removed. If all the components are the same but purity annotations are different, we replace the constraint with $a_1 \ne a_2 \Rightarrow a = \mathsf{i}$. Note that more than one such constraint could be produced, if component types are again function types and are the same except for purity annotations.

So far, the transformation of constraints preserves the solution of the constraints: the possible annotations are the same before and after the transformation. The situation becomes different when one or both of the types are type variables. Unless the two types are the same type variable (in which case, the constraint is removed), it is difficult to perform the transformation. To see the difficulty, consider the following example:

$$\mathsf{let}\ f = \lambda g.\, g @ \mathsf{true}\ \mathsf{in}\ f @ (\lambda x.\, x)$$

The typing derivation for $\lambda g.\, g @ \mathsf{true}$ is shown in Figure 6 together with collected constraints. Since the type variables α_1, α_3, and α_4 do not appear in Γ, they will be generalized when $\lambda g.\, g @ \mathsf{true}$ is bound to f in the outer let expression.

At this point, consider the constraint $\alpha_3 \ne \alpha_4 \Rightarrow a_3 = \mathsf{i}$. It is not clear if α_3 and α_4 are the same type or not. In the above example, since f is applied to $\lambda x.\, x$, we could equate α_3 and α_4 so that a_3 would become p. However, if f is applied to an impure function, such as $\lambda x.\, Sk.\, x$, α_3 and α_4 are different and we have to set a_3 to i. In other words, the purity annotation of the body of f depends on how it is used. This contradicts to the way the standard type inference algorithm for let-polymorphic languages goes. We want to determine the type of f once and for all before examining how it is used later.

In this paper, we treat all the type variables different and do not pursue the possibility of obtaining a better annotation by instantiating type variables. In the above example, we regard $\alpha_3 \ne \alpha_4$ and assign i to both a_1 (and hence a_2 via $a_1 \le a_2$) and a_3, making f completely impure.

With this design choice, we can separate the type inference into two phases: the standard type inference for let-polymorphic languages and constraint solving. Since the latter never instantiate type variables, we never have to redo the type inference. On the other hand, this choice sometimes throw away better annotations if we employed more thorough search.

$$\begin{array}{lcl}
C \cup \{a = \mathsf{i}\} & \Rightarrow & C[\mathsf{i}/a] \\
C \cup \{a = \mathsf{p}\} & \Rightarrow & C[\mathsf{p}/a] \\
 & & \\
C \cup \{\mathsf{p} \le \mathsf{p}\} & \Rightarrow & C \\
C \cup \{\mathsf{p} \le \mathsf{i}\} & \Rightarrow & C \\
C \cup \{\mathsf{p} \le a_2\} & \Rightarrow & C \\
C \cup \{\mathsf{i} \le \mathsf{p}\} & \Rightarrow & \text{type error} \\
C \cup \{\mathsf{i} \le \mathsf{i}\} & \Rightarrow & C \\
C \cup \{\mathsf{i} \le a_2\} & \Rightarrow & C \cup \{a_2 = \mathsf{i}\} \\
C \cup \{a_1 \le \mathsf{p}\} & \Rightarrow & C \cup \{a_1 = \mathsf{p}\} \\
C \cup \{a_1 \le \mathsf{i}\} & \Rightarrow & C \\
C \cup \{a_1 \le a_2\} & & \text{(non-trivial constraint)}
\end{array}$$

$$\begin{array}{lcl}
C \cup \{a_1 \ne a_2 \Rightarrow \mathsf{p} = \mathsf{i}\} & \Rightarrow & C \cup \{a_1 \le a_2, a_2 \le a_1\} \\
C \cup \{a_1 \ne a_2 \Rightarrow \mathsf{i} = \mathsf{i}\} & \Rightarrow & C \\
C \cup \{\mathsf{p} \ne \mathsf{p} \Rightarrow a = \mathsf{i}\} & \Rightarrow & C \\
C \cup \{\mathsf{p} \ne \mathsf{i} \Rightarrow a = \mathsf{i}\} & \Rightarrow & C \cup \{a = \mathsf{i}\} \\
C \cup \{\mathsf{p} \ne a_2 \Rightarrow a = \mathsf{i}\} & \Rightarrow & C \cup \{a_2 \ne \mathsf{p} \Rightarrow a = \mathsf{i}\} \\
C \cup \{\mathsf{i} \ne \mathsf{p} \Rightarrow a = \mathsf{i}\} & \Rightarrow & C \cup \{a = \mathsf{i}\} \\
C \cup \{\mathsf{i} \ne \mathsf{i} \Rightarrow a = \mathsf{i}\} & \Rightarrow & C \\
C \cup \{\mathsf{i} \ne a_2 \Rightarrow a = \mathsf{i}\} & \Rightarrow & C \cup \{a_2 \ne \mathsf{i} \Rightarrow a = \mathsf{i}\} \\
C \cup \{a_1 \ne \mathsf{p} \Rightarrow a = \mathsf{i}\} & & \text{(non-trivial constraint)} \\
C \cup \{a_1 \ne \mathsf{i} \Rightarrow a = \mathsf{i}\} & & \text{(non-trivial constraint)} \\
C \cup \{a_1 \ne a_2 \Rightarrow a = \mathsf{i}\} & & \text{(non-trivial constraint)}
\end{array}$$

Figure 7. Solving trivial constraints

4.2 Solving Trivial Constraints

Once all the constraints are collected and are transformed into the form that does not mention types, we solve trivial constraints. Let C be the set of constraints. We apply the rules $C \Rightarrow C'$ shown in Figure 7 repeatedly until no rules are applicable.

The rules are self-explanatory. The notation $C[a'/a]$ represents C where all the occurrence of a is replaced with a'. The constraints marked as "non-trivial constraint" are not handled at this point.

The constraint $\mathsf{i} \le \mathsf{p}$ usually does not arise, since no syntactic constructs (except for syntactically pure terms) enforce an annotation to be p. However, it arises when we introduce constants (such as library functions) that do enforce its argument to be pure. See Section 8.

It is easy to see that the transformation in Figure 7 is terminating and preserves solutions.

4.3 Solving Remaining Constraints

As a result of solving trivial constraints, we are left with the following forms of constraints that are marked as non-trivial constraints in Figure 7:

$$a_1 \neq \mathsf{p} \Rightarrow a = \mathsf{i} \quad a_1 \neq \mathsf{i} \Rightarrow a = \mathsf{i} \quad a_1 \neq a_2 \Rightarrow a = \mathsf{i}$$
$$a_1 \leq a_2$$

Among them, the first three constraints are not easy to solve. If we expand the logical implication, they are expressed as follows:

$$a_1 = \mathsf{p} \vee a = \mathsf{i} \quad a_1 = \mathsf{i} \vee a = \mathsf{i} \quad a_1 = a_2 \vee a = \mathsf{i}$$

It is not immediately clear which of the disjunction in the constraints to take. We could employ computationally heavy approaches such as an exhaustive search to find a solution. However, it is still not clear which of the obtained solutions to take, since it is known that the best solution does not exist in general (see Section 4.5). Furthermore, greedy algorithm to take p whenever possible could lead to unsatisfiable constraints. For example, suppose we have the following constraints:

$$a_1 = \mathsf{p} \vee a = \mathsf{i} \quad a_1 = \mathsf{i} \vee a_2 = \mathsf{i} \quad a_2 \leq a_1$$

If we take $a_1 = \mathsf{p}$ in the first constraint, a_2 becomes i from the second constraint. Then, the third constraint becomes unsatisfiable. Notice that the above three constraints can be satisfied if we choose $a = \mathsf{i}$, $a_1 = \mathsf{i}$, and $a_2 = \mathsf{p}$.

From these observations, we take the following safe and computationally light approach: we satisfy the remaining constraints of the form:

$$a_1 \neq \mathsf{p} \Rightarrow a = \mathsf{i} \quad a_1 \neq \mathsf{i} \Rightarrow a = \mathsf{i} \quad a_1 \neq a_2 \Rightarrow a = \mathsf{i}$$

by $a = \mathsf{i}$. In the above example, the constraints are satisfied by choosing $a = \mathsf{i}$, $a_1 = \mathsf{i}$, and $a_2 = \mathsf{i}$. With this design choice, we could miss a better solution: in the above example, we could assign p to a_2. We could come up with an example term that actually produces such constraints. However, the term is quite artificial. It is a future work to investigate whether employing computationally heavy approach here would pay off in practice.

4.4 Assigning p to Remaining Annotations

At this point, we are left with constraints of the form $a_1 \leq a_2$ without any further constraints. Since we want a solution that is as pure as possible, we solve them by assigning p to all the remaining annotations.

4.5 An Example Term That Has No Best Annotation

Assume we have addition on integers. Here is an example term that does not have the best annotation:[6]

$$\lambda f.\, \lambda g.\, \langle f \mathbin{@} 1 + g \mathbin{@} 2 \rangle = \mathsf{true}$$

[6] The example is due to Kanae Tsushima.

Given f and g, the function checks whether the result of adding $f \mathbin{@} 1$ and $g \mathbin{@} 2$ in a delimited context would be equal to true. Since addition always returns an integer, at least f or g is impure and change the answer type from integer to boolean. The types of f and g becomes as follows:

$$\begin{array}{lcll} f & : & \mathsf{int} \to \mathsf{int}@\mathsf{cps}[\tau_1, \mathsf{bool}, a_1] & \tau_1 \neq \mathsf{bool} \Rightarrow a_1 = \mathsf{i} \\ g & : & \mathsf{int} \to \mathsf{int}@\mathsf{cps}[\mathsf{int}, \tau_1, a_2] & \mathsf{int} \neq \tau_1 \Rightarrow a_2 = \mathsf{i} \end{array}$$

Our constraint solver would assign i to both a_1 and a_2. However, to type check the example term, we do not have to assign i to both: one of them suffices. Depending on which of the two arguments, f or g, is impure, we obtain two incomparable locally best solutions: $a_1 = \mathsf{i}$, $a_2 = \mathsf{p}$ and $a_1 = \mathsf{p}$, $a_2 = \mathsf{i}$.

This example also shows that our constraint solver does not even produce a locally best solution. Both the two incomparable solutions are better than $a_1 = \mathsf{i}$, $a_2 = \mathsf{i}$.

5 Selective CPS Transformation

In this section, we show the selective CPS transformation that transforms impure parts of a term e into CPS while pure parts of e is kept as is in direct style. The transformation is in one pass [8]: we perform the reduction of administrative β-redexes at transformation time.

The (call-by-value, left-to-right) selective CPS transformation is shown in Figure 8. It receives an annotated term e^a and dispatches over the purity annotation a first and then over e. The output of the selective CPS transformation is expressed in a two-level language. The overlined constructs are *static* and are reduced at transformation time. The underlined constructs are *dynamic* and represent syntactic constructors. Since static constructs are reduced at transformation time, the result of selective CPS transformation becomes a completely dynamic term. We assume that bound variables in dynamic λ-abstraction, fixed points, and let expressions are chosen fresh.

A pure term e^{p} is transformed using $[\![e^{\mathsf{p}}]\!]_{\mathsf{p}}$. If all the subterms of e^{p} are pure, $[\![e^{\mathsf{p}}]\!]_{\mathsf{p}}$ is an identity function: it returns the input term e^{p} with all the purity annotations removed and all the syntactic constructors underlined. In particular, the result is in direct style, not in CPS.

If a part of a term is annotated as impure, that part is transformed into CPS. For example, $(\lambda^{\mathsf{i}} x.\, {e_1}^{\mathsf{p}})^{\mathsf{p}}$ is transformed to $\underline{\lambda} x.\, \underline{\lambda} k.\, k \mathbin{\underline{@}} [\![{e_1}^{\mathsf{p}}]\!]_{\mathsf{p}}$. Since the λ-abstraction is annotated as impure, the result receives a continuation k, which is then applied to $[\![{e_1}^{\mathsf{p}}]\!]_{\mathsf{p}}$. If the body e_1 is also impure, as in $(\lambda^{\mathsf{i}} x.\, {e_1}^{\mathsf{i}})^{\mathsf{p}}$, the body is transformed into CPS, as we explain next.

An impure term e^{i} is transformed using $[\![e^{\mathsf{i}}]\!]_{\mathsf{i}}$. The impure transformation $[\![e^{\mathsf{i}}]\!]_{\mathsf{i}}$ additionally receives a static continuation k as a transformation-time argument. By receiving the continuation k as a transformation-time function rather than a dynamic term, it becomes possible to reduce administrative β-redexes at transformation time.

$$
\begin{array}{rcl}
\llbracket c^{\mathsf{p}} \rrbracket_{\mathsf{p}} & = & c \\
\llbracket x^{\mathsf{p}} \rrbracket_{\mathsf{p}} & = & x \\
\llbracket (\lambda^{\mathsf{p}} x.\, e_1{}^{\mathsf{p}})^{\mathsf{p}} \rrbracket_{\mathsf{p}} & = & \underline{\lambda} x.\, \llbracket e_1{}^{\mathsf{p}} \rrbracket_{\mathsf{p}} \\
\llbracket (\lambda^{\mathsf{i}} x.\, e_1{}^{\mathsf{p}})^{\mathsf{p}} \rrbracket_{\mathsf{p}} & = & \underline{\lambda} x.\, \underline{\lambda} k.\, k \mathbin{\underline{@}} \llbracket e_1{}^{\mathsf{p}} \rrbracket_{\mathsf{p}} \\
\llbracket (\lambda^{\mathsf{i}} x.\, e_1{}^{\mathsf{i}})^{\mathsf{p}} \rrbracket_{\mathsf{p}} & = & \underline{\lambda} x.\, \underline{\lambda} k.\, \llbracket e_1{}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \mathbin{\overline{@}} (\overline{\lambda} v.\, k \mathbin{\underline{@}} v) \\
\llbracket (\mathsf{fix}^{\mathsf{p}}\, f.x.\, e_1{}^{\mathsf{p}})^{\mathsf{p}} \rrbracket_{\mathsf{p}} & = & \underline{\mathsf{fix}}\, f.x.\, \llbracket e_1{}^{\mathsf{p}} \rrbracket_{\mathsf{p}} \\
\llbracket (\mathsf{fix}^{\mathsf{i}}\, f.x.\, e_1{}^{\mathsf{p}})^{\mathsf{p}} \rrbracket_{\mathsf{p}} & = & \underline{\lambda} x.\, \underline{\lambda} k.\, k \mathbin{\underline{@}} ((\underline{\mathsf{fix}}\, f.x.\, \llbracket e_1{}^{\mathsf{p}} \rrbracket_{\mathsf{p}}) \mathbin{\underline{@}} x) \\
\llbracket (\mathsf{fix}^{\mathsf{i}}\, f.x.\, e_1{}^{\mathsf{i}})^{\mathsf{p}} \rrbracket_{\mathsf{p}} & = & \underline{\mathsf{fix}}\, f.x.\, \underline{\lambda} k.\, \llbracket e_1{}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \mathbin{\overline{@}} (\overline{\lambda} v.\, k \mathbin{\underline{@}} v) \\
\llbracket (e_1{}^{\mathsf{p}} \mathbin{@^{\mathsf{p}}} e_2{}^{\mathsf{p}})^{\mathsf{p}} \rrbracket_{\mathsf{p}} & = & \llbracket e_1{}^{\mathsf{p}} \rrbracket_{\mathsf{p}} \mathbin{\underline{@}} \llbracket e_2{}^{\mathsf{p}} \rrbracket_{\mathsf{p}} \\
\llbracket \langle e_1{}^{\mathsf{p}} \rangle^{\mathsf{p}} \rrbracket_{\mathsf{p}} & = & \llbracket e_1{}^{\mathsf{p}} \rrbracket_{\mathsf{p}} \\
\llbracket \langle e_1{}^{\mathsf{i}} \rangle^{\mathsf{p}} \rrbracket_{\mathsf{p}} & = & \llbracket e_1{}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \mathbin{\overline{@}} (\overline{\lambda} v.\, v) \\
\llbracket (\mathsf{let}\, x = v_1{}^{\mathsf{p}}\, \mathsf{in}\, e_2{}^{\mathsf{p}})^{\mathsf{p}} \rrbracket_{\mathsf{p}} & = & \underline{\mathsf{let}}\, x = \llbracket v_1{}^{\mathsf{p}} \rrbracket_{\mathsf{p}}\, \underline{\mathsf{in}}\, \llbracket e_2{}^{\mathsf{p}} \rrbracket_{\mathsf{p}} \\
\llbracket (\mathsf{if}\, e_1{}^{\mathsf{p}}\, \mathsf{then}\, e_2{}^{\mathsf{p}}\, \mathsf{else}\, e_3{}^{\mathsf{p}})^{\mathsf{p}} \rrbracket_{\mathsf{p}} & = & \underline{\mathsf{if}}\, \llbracket e_1{}^{\mathsf{p}} \rrbracket_{\mathsf{p}}\, \underline{\mathsf{then}}\, \llbracket e_2{}^{\mathsf{p}} \rrbracket_{\mathsf{p}}\, \underline{\mathsf{else}}\, \llbracket e_3{}^{\mathsf{p}} \rrbracket_{\mathsf{p}} \\
\\
\llbracket (e_1{}^{\mathsf{p}} \mathbin{@^{\mathsf{p}}} e_2{}^{\mathsf{p}})^{\mathsf{i}} \rrbracket_{\mathsf{i}} & = & \overline{\lambda} k.\, (\underline{\lambda} v.\, k \mathbin{\overline{@}} v) \mathbin{\underline{@}} (\llbracket e_1{}^{\mathsf{p}} \rrbracket_{\mathsf{p}} \mathbin{\underline{@}} \llbracket e_2{}^{\mathsf{p}} \rrbracket_{\mathsf{p}}) \\
\llbracket (e_1{}^{\mathsf{p}} \mathbin{@^{\mathsf{p}}} e_2{}^{\mathsf{i}})^{\mathsf{i}} \rrbracket_{\mathsf{i}} & = & \overline{\lambda} k.\, (\underline{\lambda} v_1.\, \llbracket e_2{}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \mathbin{\overline{@}} (\overline{\lambda} v_2.\, (\underline{\lambda} v.\, k \mathbin{\overline{@}} v) \mathbin{\underline{@}} (v_1 \mathbin{\underline{@}} v_2))) \mathbin{\underline{@}} \llbracket e_1{}^{\mathsf{p}} \rrbracket_{\mathsf{p}} \\
\llbracket (e_1{}^{\mathsf{i}} \mathbin{@^{\mathsf{p}}} e_2{}^{\mathsf{p}})^{\mathsf{i}} \rrbracket_{\mathsf{i}} & = & \overline{\lambda} k.\, \llbracket e_1{}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \mathbin{\overline{@}} (\overline{\lambda} v_1.\, (\underline{\lambda} v.\, k \mathbin{\overline{@}} v) \mathbin{\underline{@}} (v_1 \mathbin{\underline{@}} \llbracket e_2{}^{\mathsf{p}} \rrbracket_{\mathsf{p}})) \\
\llbracket (e_1{}^{\mathsf{i}} \mathbin{@^{\mathsf{p}}} e_2{}^{\mathsf{i}})^{\mathsf{i}} \rrbracket_{\mathsf{i}} & = & \overline{\lambda} k.\, \llbracket e_1{}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \mathbin{\overline{@}} (\overline{\lambda} v_1.\, \llbracket e_2{}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \mathbin{\overline{@}} (\overline{\lambda} v_2.\, (\underline{\lambda} v.\, k \mathbin{\overline{@}} v) \mathbin{\underline{@}} (v_1 \mathbin{\underline{@}} v_2))) \\
\llbracket (e_1{}^{\mathsf{p}} \mathbin{@^{\mathsf{i}}} e_2{}^{\mathsf{p}})^{\mathsf{i}} \rrbracket_{\mathsf{i}} & = & \overline{\lambda} k.\, (\llbracket e_1{}^{\mathsf{p}} \rrbracket_{\mathsf{p}} \mathbin{\underline{@}} \llbracket e_2{}^{\mathsf{p}} \rrbracket_{\mathsf{p}}) \mathbin{\underline{@}} (\underline{\lambda} v.\, k \mathbin{\overline{@}} v) \\
\llbracket (e_1{}^{\mathsf{p}} \mathbin{@^{\mathsf{i}}} e_2{}^{\mathsf{i}})^{\mathsf{i}} \rrbracket_{\mathsf{i}} & = & \overline{\lambda} k.\, (\underline{\lambda} v_1.\, \llbracket e_2{}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \mathbin{\overline{@}} (\overline{\lambda} v_2.\, (v_1 \mathbin{\underline{@}} v_2) \mathbin{\underline{@}} (\underline{\lambda} v.\, k \mathbin{\overline{@}} v))) \mathbin{\underline{@}} \llbracket e_1{}^{\mathsf{p}} \rrbracket_{\mathsf{p}} \\
\llbracket (e_1{}^{\mathsf{i}} \mathbin{@^{\mathsf{i}}} e_2{}^{\mathsf{p}})^{\mathsf{i}} \rrbracket_{\mathsf{i}} & = & \overline{\lambda} k.\, \llbracket e_1{}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \mathbin{\overline{@}} (\overline{\lambda} v_1.\, (v_1 \mathbin{\underline{@}} \llbracket e_2{}^{\mathsf{p}} \rrbracket_{\mathsf{p}}) \mathbin{\underline{@}} (\underline{\lambda} v.\, k \mathbin{\overline{@}} v)) \\
\llbracket (e_1{}^{\mathsf{i}} \mathbin{@^{\mathsf{i}}} e_2{}^{\mathsf{i}})^{\mathsf{i}} \rrbracket_{\mathsf{i}} & = & \overline{\lambda} k.\, \llbracket e_1{}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \mathbin{\overline{@}} (\overline{\lambda} v_1.\, \llbracket e_2{}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \mathbin{\overline{@}} (\overline{\lambda} v_2.\, (v_1 \mathbin{\underline{@}} v_2) \mathbin{\underline{@}} (\underline{\lambda} v.\, k \mathbin{\overline{@}} v))) \\
\llbracket (S^{\mathsf{p}} x.\, e_1{}^{\mathsf{p}})^{\mathsf{i}} \rrbracket_{\mathsf{i}} & = & \overline{\lambda} k.\, \underline{\mathsf{let}}\, x = \underline{\lambda} v.\, k \mathbin{\overline{@}} v\, \underline{\mathsf{in}}\, \llbracket e_1{}^{\mathsf{p}} \rrbracket_{\mathsf{p}} \\
\llbracket (S^{\mathsf{p}} x.\, e_1{}^{\mathsf{i}})^{\mathsf{i}} \rrbracket_{\mathsf{i}} & = & \overline{\lambda} k.\, \underline{\mathsf{let}}\, x = \underline{\lambda} v.\, k \mathbin{\overline{@}} v\, \underline{\mathsf{in}}\, \llbracket e_1{}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \mathbin{\overline{@}} (\overline{\lambda} v.\, v) \\
\llbracket (S^{\mathsf{i}} x.\, e_1{}^{\mathsf{p}})^{\mathsf{i}} \rrbracket_{\mathsf{i}} & = & \overline{\lambda} k.\, \underline{\mathsf{let}}\, x = \underline{\lambda} v.\, \underline{\lambda} k'.\, k' \mathbin{\underline{@}} (k \mathbin{\overline{@}} v)\, \underline{\mathsf{in}}\, \llbracket e_1{}^{\mathsf{p}} \rrbracket_{\mathsf{p}} \\
\llbracket (S^{\mathsf{i}} x.\, e_1{}^{\mathsf{i}})^{\mathsf{i}} \rrbracket_{\mathsf{i}} & = & \overline{\lambda} k.\, \underline{\mathsf{let}}\, x = \underline{\lambda} v.\, \underline{\lambda} k'.\, k' \mathbin{\underline{@}} (k \mathbin{\overline{@}} v)\, \underline{\mathsf{in}}\, \llbracket e_1{}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \mathbin{\overline{@}} (\overline{\lambda} v.\, v) \\
\llbracket (\mathsf{let}\, x = v_1{}^{\mathsf{p}}\, \mathsf{in}\, e_2{}^{\mathsf{i}})^{\mathsf{i}} \rrbracket_{\mathsf{i}} & = & \overline{\lambda} k.\, \underline{\mathsf{let}}\, x = \llbracket v_1{}^{\mathsf{p}} \rrbracket_{\mathsf{p}}\, \underline{\mathsf{in}}\, \llbracket e_2{}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \mathbin{\overline{@}} k \\
\llbracket (\mathsf{if}\, e_1{}^{\mathsf{p}}\, \mathsf{then}\, e_2{}^{\mathsf{p}}\, \mathsf{else}\, e_3{}^{\mathsf{p}})^{\mathsf{i}} \rrbracket_{\mathsf{i}} & = & \overline{\lambda} k.\, (\underline{\lambda} v.\, k \mathbin{\overline{@}} v) \mathbin{\underline{@}} (\underline{\mathsf{if}}\, \llbracket e_1{}^{\mathsf{p}} \rrbracket_{\mathsf{p}}\, \underline{\mathsf{then}}\, \llbracket e_2{}^{\mathsf{p}} \rrbracket_{\mathsf{p}}\, \underline{\mathsf{else}}\, \llbracket e_3{}^{\mathsf{p}} \rrbracket_{\mathsf{p}}) \\
\llbracket (\mathsf{if}\, e_1{}^{\mathsf{p}}\, \mathsf{then}\, e_2{}^{\mathsf{p}}\, \mathsf{else}\, e_3{}^{\mathsf{i}})^{\mathsf{i}} \rrbracket_{\mathsf{i}} & = & \overline{\lambda} k.\, \underline{\mathsf{if}}\, \llbracket e_1{}^{\mathsf{p}} \rrbracket_{\mathsf{p}}\, \underline{\mathsf{then}}\, (\underline{\lambda} v.\, k \mathbin{\overline{@}} v) \mathbin{\underline{@}} \llbracket e_2{}^{\mathsf{p}} \rrbracket_{\mathsf{p}}\, \underline{\mathsf{else}}\, \llbracket e_3{}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \mathbin{\overline{@}} k \\
\llbracket (\mathsf{if}\, e_1{}^{\mathsf{p}}\, \mathsf{then}\, e_2{}^{\mathsf{i}}\, \mathsf{else}\, e_3{}^{\mathsf{p}})^{\mathsf{i}} \rrbracket_{\mathsf{i}} & = & \overline{\lambda} k.\, \underline{\mathsf{if}}\, \llbracket e_1{}^{\mathsf{p}} \rrbracket_{\mathsf{p}}\, \underline{\mathsf{then}}\, \llbracket e_2{}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \mathbin{\overline{@}} k\, \underline{\mathsf{else}}\, (\underline{\lambda} v.\, k \mathbin{\overline{@}} v) \mathbin{\underline{@}} \llbracket e_3{}^{\mathsf{p}} \rrbracket_{\mathsf{p}} \\
\llbracket (\mathsf{if}\, e_1{}^{\mathsf{p}}\, \mathsf{then}\, e_2{}^{\mathsf{i}}\, \mathsf{else}\, e_3{}^{\mathsf{i}})^{\mathsf{i}} \rrbracket_{\mathsf{i}} & = & \overline{\lambda} k.\, \underline{\mathsf{if}}\, \llbracket e_1{}^{\mathsf{p}} \rrbracket_{\mathsf{p}}\, \underline{\mathsf{then}}\, \llbracket e_2{}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \mathbin{\overline{@}} k\, \underline{\mathsf{else}}\, \llbracket e_3{}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \mathbin{\overline{@}} k \\
\llbracket (\mathsf{if}\, e_1{}^{\mathsf{i}}\, \mathsf{then}\, e_2{}^{\mathsf{p}}\, \mathsf{else}\, e_3{}^{\mathsf{p}})^{\mathsf{i}} \rrbracket_{\mathsf{i}} & = & \overline{\lambda} k.\, \llbracket e_1{}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \mathbin{\overline{@}} (\overline{\lambda} v_1.\, (\underline{\lambda} v.\, k \mathbin{\overline{@}} v) \mathbin{\underline{@}} (\underline{\mathsf{if}}\, v_1\, \underline{\mathsf{then}}\, \llbracket e_2{}^{\mathsf{p}} \rrbracket_{\mathsf{p}}\, \underline{\mathsf{else}}\, \llbracket e_3{}^{\mathsf{p}} \rrbracket_{\mathsf{p}})) \\
\llbracket (\mathsf{if}\, e_1{}^{\mathsf{i}}\, \mathsf{then}\, e_2{}^{\mathsf{p}}\, \mathsf{else}\, e_3{}^{\mathsf{i}})^{\mathsf{i}} \rrbracket_{\mathsf{i}} & = & \overline{\lambda} k.\, \llbracket e_1{}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \mathbin{\overline{@}} (\overline{\lambda} v_1.\, \underline{\mathsf{if}}\, v_1\, \underline{\mathsf{then}}\, (\underline{\lambda} v.\, k \mathbin{\overline{@}} v) \mathbin{\underline{@}} \llbracket e_2{}^{\mathsf{p}} \rrbracket_{\mathsf{p}}\, \underline{\mathsf{else}}\, \llbracket e_3{}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \mathbin{\overline{@}} k) \\
\llbracket (\mathsf{if}\, e_1{}^{\mathsf{i}}\, \mathsf{then}\, e_2{}^{\mathsf{i}}\, \mathsf{else}\, e_3{}^{\mathsf{p}})^{\mathsf{i}} \rrbracket_{\mathsf{i}} & = & \overline{\lambda} k.\, \llbracket e_1{}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \mathbin{\overline{@}} (\overline{\lambda} v_1.\, \underline{\mathsf{if}}\, v_1\, \underline{\mathsf{then}}\, \llbracket e_2{}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \mathbin{\overline{@}} k\, \underline{\mathsf{else}}\, (\underline{\lambda} v.\, k \mathbin{\overline{@}} v) \mathbin{\underline{@}} \llbracket e_3{}^{\mathsf{p}} \rrbracket_{\mathsf{p}}) \\
\llbracket (\mathsf{if}\, e_1{}^{\mathsf{i}}\, \mathsf{then}\, e_2{}^{\mathsf{i}}\, \mathsf{else}\, e_3{}^{\mathsf{i}})^{\mathsf{i}} \rrbracket_{\mathsf{i}} & = & \overline{\lambda} k.\, \llbracket e_1{}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \mathbin{\overline{@}} (\overline{\lambda} v_1.\, \underline{\mathsf{if}}\, v_1\, \underline{\mathsf{then}}\, \llbracket e_2{}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \mathbin{\overline{@}} k\, \underline{\mathsf{else}}\, \llbracket e_3{}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \mathbin{\overline{@}} k)
\end{array}
$$

Figure 8. Selective CPS transformation for annotated terms

Let us go back to the transformation of $(\lambda^{\mathsf{i}} x.\, e_1{}^{\mathsf{i}})^{\mathsf{p}}$. Its CPS transformation is $\underline{\lambda} x.\, \underline{\lambda} k.\, \llbracket e_1{}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \mathbin{\overline{@}} (\overline{\lambda} v.\, k \mathbin{\underline{@}} v)$. The body e_1 is transformed with a static continuation $\overline{\lambda} v.\, k \mathbin{\underline{@}} v$, the two-level η-expansion of k. When applied to an argument v at transformation time, it returns a dynamic application $k \mathbin{\underline{@}} v$. This two-level η-expansion in the one-pass CPS transformation results in an administrative η-redex [8]. We can avoid creation of administrative η-redexes by duplicating $\llbracket e^{\mathsf{i}} \rrbracket_{\mathsf{i}}$ into two: one for the tail case and the other for non-tail case. See [8] for details.

The selective CPS transformation of fix is similar in spirit, but rather complex in the two impure cases. Both receive a continuation k since they are impure externally. However, they differ in if the generated recursive function receives a continuation or not. When the body e_1 is pure, the generated recursive function $(\underline{\mathsf{fix}}\, f.x.\, \llbracket e_1{}^{\mathsf{p}} \rrbracket_{\mathsf{p}})$ is in direct style and is applied to the argument x, whose result is passed to k. It is incorrect to receive a continuation inside fix as in $\underline{\mathsf{fix}}\, f.x.\, \underline{\lambda} k.\, k \mathbin{\underline{@}} \llbracket e_1{}^{\mathsf{p}} \rrbracket_{\mathsf{p}}$, because then f in e_1 would have to receive a continuation. The latter would not satisfy preservation of types (Theorem 5.1) shown later.

$$
\begin{array}{lcll}
E & ::= & [\,] \mid (E \mathbin{@^{a_3}} {e_2}^{a_2})^a \mid ({v_1}^{\mathsf{p}} \mathbin{@^{a_3}} E)^a \mid (\mathsf{if}\,E\,\mathsf{then}\,{e_2}^{a_2}\,\mathsf{else}\,{e_3}^{a_3})^a \mid \langle E\rangle^{\mathsf{p}} & (a_i \le a \text{ for all } i) \quad \text{evaluation context}\\
F & ::= & [\,] \mid (F \mathbin{@^{a_3}} {e_2}^{a_2})^a \mid ({v_1}^{\mathsf{p}} \mathbin{@^{a_3}} F)^a \mid (\mathsf{if}\,F\,\mathsf{then}\,{e_2}^{a_2}\,\mathsf{else}\,{e_3}^{a_3})^a & (a_i \le a \text{ for all } i) \quad \text{pure evaluation context}
\end{array}
$$

$$
\begin{array}{rcll}
((\lambda^{a_3}x.\,{e_1}^{a_1})^{\mathsf{p}} \mathbin{@^{a_3}} {v_2}^{\mathsf{p}})^a & \rightsquigarrow & (e_1[{v_2}^{\mathsf{p}}/x])^{a_1} & a_1 \le a_3 \le a\\
((\mathsf{fix}^{a_3}\,f.x.\,{e_1}^{a_1})^{\mathsf{p}} \mathbin{@^{a_3}} {v_2}^{\mathsf{p}})^a & \rightsquigarrow & (e_1[(\mathsf{fix}^{a_1}\,f.x.\,{e_1}^{a_1})^{\mathsf{p}}/f][{v_2}^{\mathsf{p}}/x])^{a_1} & a_1 \le a_3 \le a\\
\langle {v_1}^{\mathsf{p}}\rangle^{\mathsf{p}} & \rightsquigarrow & {v_1}^{\mathsf{p}} & \\
\langle F[(\mathcal{S}^{a_2}k.\,{e_1}^{a_1})^{\mathsf{i}}]\rangle^{\mathsf{p}} & \rightsquigarrow & \langle(\mathsf{let}\,k = (\lambda^{a_2}x.\,\langle F[x^{\mathsf{p}}]\rangle^{\mathsf{p}})^{\mathsf{p}}\,\mathsf{in}\,{e_1}^{a_1})^{a_1}\rangle^{\mathsf{p}} & \\
(\mathsf{let}\,x = {v_1}^{\mathsf{p}}\,\mathsf{in}\,{e_2}^{a_2})^{a_2} & \rightsquigarrow & (e_2[{v_1}^{\mathsf{p}}/x])^{a_2} & a_2 \le a\\
(\mathsf{if}\,\mathsf{true}^{\mathsf{p}}\,\mathsf{then}\,{e_2}^{a_2}\,\mathsf{else}\,{e_3}^{a_3})^a & \rightsquigarrow & {e_2}^{a_2} & a_2 \le a \quad a_3 \le a\\
(\mathsf{if}\,\mathsf{false}^{\mathsf{p}}\,\mathsf{then}\,{e_2}^{a_2}\,\mathsf{else}\,{e_3}^{a_3})^a & \rightsquigarrow & {e_3}^{a_3} & a_2 \le a \quad a_3 \le a
\end{array}
$$

$$
\left.
\begin{array}{cc}
\dfrac{{e_1}^{a_1} \rightsquigarrow {e'_1}^{a'_1}}{([{e_1}^{a_1}] \mathbin{@^{a_3}} {e_2}^{a_2})^a \rightsquigarrow ([{e'_1}^{a'_1}] \mathbin{@^{a_3}} {e_2}^{a_2})^a}
&
\dfrac{{e_2}^{a_2} \rightsquigarrow {e'_2}^{a'_2}}{({v_1}^{\mathsf{p}} \mathbin{@^{a_3}} [{e_2}^{a_2}])^a \rightsquigarrow ({v_1}^{\mathsf{p}} \mathbin{@^{a_3}} [{e'_2}^{a'_2}])^a}
\\[2ex]
\dfrac{{e_1}^{a_1} \rightsquigarrow {e'_1}^{a'_1}}{(\mathsf{if}\,[{e_1}^{a_1}]\,\mathsf{then}\,{e_2}^{a_2}\,\mathsf{else}\,{e_3}^{a_3})^a \rightsquigarrow (\mathsf{if}\,[{e'_1}^{a'_1}]\,\mathsf{then}\,{e_2}^{a_2}\,\mathsf{else}\,{e_3}^{a_3})^a}
&
\dfrac{{e_1}^{a_1} \rightsquigarrow {e'_1}^{a'_1}}{\langle[{e_1}^{a_1}]\rangle^{\mathsf{p}} \rightsquigarrow \langle[{e'_1}^{a'_1}]\rangle^{\mathsf{p}}}
\end{array}
\right\}\ a_i \le a \text{ (and } a'_i \le a_i\text{) for all } i
$$

Figure 9. Reduction rules

When the body e_1 is impure, on the other hand, the generated recursive function is in CPS. This is witnessed by observing the generated function $\underline{\mathsf{fix}}\,f.x.\,\underline{\lambda}k.\,\llbracket {e_1}^{\mathsf{i}}\rrbracket_{\mathsf{i}} \mathbin{\overline{@}} (\overline{\lambda}v.\,k \mathbin{\underline{@}} v)$ receives a continuation inside fix. Similarly to the previous case, it is incorrect to receive a continuation outside fix as in $\underline{\lambda}x.\,\underline{\lambda}k.\,((\underline{\mathsf{fix}}\,f.x.\,\llbracket {e_1}^{\mathsf{i}}\rrbracket_{\mathsf{i}} \mathbin{\overline{@}} (\overline{\lambda}v.\,k \mathbin{\underline{@}} v)) \mathbin{\underline{@}} x)$ because then f in e_1 would be in direct style.

The other pure rules are straightforward. For $\langle {e_1}^{\mathsf{i}}\rangle$, we supply an empty continuation $(\overline{\lambda}v.\,v)$ to transform the impure body ${e_1}^{\mathsf{i}}$.

Impure rules are split into many cases according to the value of purity annotations of subterms. Basically, when we transform an impure subterm e^{i}, we perform the standard CPS transformation, while when we transform a pure subterm e^{p}, we construct a dynamic application $(\underline{\lambda}v.\,k \mathbin{\overline{@}} v) \mathbin{\underline{@}} \llbracket e^{\mathsf{p}}\rrbracket_{\mathsf{p}}$. It is incorrect to reduce this dynamic application at transformation time as in $k \mathbin{\overline{@}} \llbracket e^{\mathsf{p}}\rrbracket_{\mathsf{p}}$, because k might place its argument under dynamic abstraction, changing the order of evaluation.

There are eight[7] impure rules for applications, each case corresponding to the value of the three purity annotations. Mostly, they follow the basic strategy described above. In addition, we have to be careful not to change the order of evaluation. In the second rule for application, $({e_1}^{\mathsf{p}} \mathbin{@^{\mathsf{p}}} {e_2}^{\mathsf{i}})^{\mathsf{i}}$, we construct a dynamic application of the form

$$(\underline{\lambda}v_1.\,\llbracket {e_2}^{\mathsf{i}}\rrbracket_{\mathsf{i}} \mathbin{\overline{@}} (\overline{\lambda}v_2.\ldots)) \mathbin{\underline{@}} \llbracket {e_1}^{\mathsf{p}}\rrbracket_{\mathsf{p}}$$

It is incorrect to perform the dynamic application at transformation time and inline $\llbracket {e_1}^{\mathsf{p}}\rrbracket_{\mathsf{p}}$, because ${e_1}^{\mathsf{p}}$ would then be evaluated after ${e_2}^{\mathsf{i}}$ at runtime. The same for the case $({e_1}^{\mathsf{p}} \mathbin{@^{\mathsf{i}}} {e_2}^{\mathsf{i}})^{\mathsf{i}}$.

[7] Since we have one pure rule, there are nine rules for applications in total. The same for conditionals.

The rule for shift is split into four cases, according to the purity of the captured continuation and the body of shift. In all the cases, a dynamic let expression is used for the captured continuation to retain the polymorphism of the continuation's answer type.

The rules for let expressions and conditionals are straightforward, following the basic strategy.

Define the CPS transformation of types as follows:

$$
\begin{array}{rcl}
\alpha^* & = & \alpha\\
b^* & = & b\\
(\tau_2 \to \tau_1 @\mathrm{cps}[\tau_3, \tau_4, \mathsf{p}])^* & = & {\tau_2}^* \to {\tau_1}^*\\
(\tau_2 \to \tau_1 @\mathrm{cps}[\tau_3, \tau_4, \mathsf{i}])^* & = & {\tau_2}^* \to ({\tau_1}^* \to {\tau_3}^*) \to {\tau_4}^*
\end{array}
$$

We can then prove that the selective CPS transformation preserves types.

Theorem 5.1 (preservation of types).
(1) If $\Gamma \vdash e^{\mathsf{p}} : \tau_1 \mathbin{@}\mathrm{cps}[\tau_2, \tau_2, \mathsf{p}]$*, then* $\Gamma^* \vdash \llbracket e^{\mathsf{p}}\rrbracket_{\mathsf{p}} : {\tau_1}^*$. *(2) If* $\Gamma \vdash e^{\mathsf{i}} : \tau_1 \mathbin{@}\mathrm{cps}[\tau_2, \tau_3, \mathsf{i}]$*, then* $\Gamma^* \vdash \underline{\lambda}k.\,\llbracket e^{\mathsf{i}}\rrbracket_{\mathsf{i}} \mathbin{\overline{@}} (\overline{\lambda}v.\,k \mathbin{\underline{@}} v) : ({\tau_1}^* \to {\tau_2}^*) \to {\tau_3}^*$.

Proof. By simultaneous induction on the derivation of $\Gamma \vdash e^{\mathsf{p}} : \tau_1 \mathbin{@}\mathrm{cps}[\tau_2, \tau_2, \mathsf{p}]$ and $\Gamma \vdash e^{\mathsf{i}} : \tau_1 \mathbin{@}\mathrm{cps}[\tau_2, \tau_3, \mathsf{i}]$ □

6 Correctness

In this section, we prove the correctness of our selective CPS transformation. We first define the reduction rules for the annotated terms. We then show that the selective CPS transformation preserves β-equality: if ${e_1}^{a_1}$ reduces to ${e_2}^{a_2}$, then their selective CPS transformations are β-equal.

Evaluation contexts, pure evaluation contexts, and reduction rules are shown in Figure 9. They are all standard, following Asai and Kameyama [2], except for the explicit purity annotations. All the annotations come with necessary inequality constraints. Four inference rules are equivalent to the following one:

$$\frac{{e_1}^{a_1} \rightsquigarrow {e'_1}^{a'_1}}{(E[{e_1}^{a_1}])^a \rightsquigarrow (E[{e'_1}^{a'_1}])^a}$$

In Figure 9, we expand the definition of evaluation contexts, because proofs of theorems become somewhat simpler.

In the reduction rules, annotations are mostly inherited from the original term. However, there are a few cases when the annotation becomes smaller.

Proposition 6.1. *If ${e_1}^{a_1} \rightsquigarrow {e_2}^{a_2}$, then $a_2 \leq a_1$.*

Proof. By inspecting all the reduction rules. Note that $E[{e_1}^{a_1}]$ and $E[{e_2}^{a_2}]$ have the same annotation when $E \neq [\,]$. □

The annotation becomes smaller in three cases. First, β-reduction replaces the topmost annotation with the annotation of the body of the function. Secondly, conditionals replaces the topmost annotation with the annotation of one of the branches. However, these cases simply inherit the annotation of the original term. The essential case is in the shift case. Before shift is reduced, the hole of the pure evaluation context F was filled with shift, which had annotation i. After the reduction, it is filled with a variable x (as the continuation capturing is done), which has annotation p. That is, the annotation of the hole of F has decreased.

We can prove progress and preservation in a standard way [2]. Even if a term is well typed, it can be a stuck term if shift occurs in an evaluation context without being surrounded by reset.

Theorem 6.2 (progress)**.** (1) *If $\Gamma \vdash {e_1}^{\mathsf{p}} : \tau_1 \,@\mathrm{cps}[\tau_2, \tau_2, \mathsf{p}]$, then either ${e_1}^{\mathsf{p}}$ is a value, or ${e_1}^{\mathsf{p}} \rightsquigarrow {e_2}^{\mathsf{p}}$ for some ${e_2}^{\mathsf{p}}$.* (2) *If $\Gamma \vdash {e_1}^{\mathsf{i}} : \tau_1 \,@\mathrm{cps}[\tau_2, \tau_3, \mathsf{i}]$, then either ${e_1}^{\mathsf{i}}$ is of the form $F[(\mathcal{S}^a k.\, {e_2}^{a_2})^{\mathsf{i}}]$ for some F, a, k, and ${e_2}^{a_2}$, or ${e_1}^{\mathsf{i}} \rightsquigarrow {e_2}^{a_2}$ for some ${e_2}^{a_2}$.*

Proof. By simultaneous induction on the derivation of $\Gamma \vdash {e_1}^{\mathsf{p}} : \tau_1 \,@\mathrm{cps}[\tau_2, \tau_2, \mathsf{p}]$ and $\Gamma \vdash {e_1}^{\mathsf{i}} : \tau_1 \,@\mathrm{cps}[\tau_2, \tau_3, \mathsf{i}]$. □

Preservation is divided into three cases, according to the purity annotations of reduced terms: ${e_1}^{\mathsf{p}} \rightsquigarrow {e_2}^{\mathsf{p}}$, ${e_1}^{\mathsf{i}} \rightsquigarrow {e_2}^{\mathsf{p}}$, or ${e_1}^{\mathsf{i}} \rightsquigarrow {e_2}^{\mathsf{i}}$.

Theorem 6.3 (preservation)**.** (1) *If $\Gamma \vdash {e_1}^{\mathsf{p}} : \tau_1 \,@\mathrm{cps}[\tau_2, \tau_2, \mathsf{p}]$ and ${e_1}^{\mathsf{p}} \rightsquigarrow {e_2}^{\mathsf{p}}$, then $\Gamma \vdash {e_2}^{\mathsf{p}} : \tau_1 \,@\mathrm{cps}[\tau_2, \tau_2, \mathsf{p}]$ (2) If $\Gamma \vdash {e_1}^{\mathsf{i}} : \tau_1 \,@\mathrm{cps}[\tau_2, \tau_3, \mathsf{i}]$ and ${e_1}^{\mathsf{i}} \rightsquigarrow {e_2}^{\mathsf{p}}$, then $\tau_2 = \tau_3$ and $\Gamma \vdash {e_2}^{\mathsf{p}} : \tau_1 \,@\mathrm{cps}[\tau_2, \tau_2, \mathsf{p}]$ (3) If $\Gamma \vdash {e_1}^{\mathsf{i}} : \tau_1 \,@\mathrm{cps}[\tau_2, \tau_3, \mathsf{i}]$ and ${e_1}^{\mathsf{i}} \rightsquigarrow {e_2}^{\mathsf{i}}$, then $\Gamma \vdash {e_2}^{\mathsf{i}} : \tau_1 \,@\mathrm{cps}[\tau_2, \tau_3, \mathsf{i}]$*

The proof is by induction on the derivation of ${e_1}^{a_1} \rightsquigarrow {e_2}^{a_2}$. As usual, we need substitution lemma [29] for β, fix, and let reductions. In addition, we need the following decomposition lemma to prove the shift case.

Lemma 6.4 (decomposition of pure context)**.**
If $\Gamma \vdash (F[{e_0}^{a_0}])^a : \tau_1 \,@\mathrm{cps}[\tau_2, \tau_3, a]$, then we have

1. *$\Gamma' \vdash {e_0}^{a_0} : \tau_0 \,@\mathrm{cps}[\tau, \tau_3, a_0]$, and*
2. *$\Gamma, x : \tau_0 \vdash (F[x^{\mathsf{p}}])^{a'} : \tau_1 \,@\mathrm{cps}[\tau_2, \tau, a']$, where x is fresh*

for some Γ', τ_0, τ, and a' such that Γ' extends Γ and $a' \leq a$.

Proof. By induction on F. □

Finally, correctness of the selective CPS transformation is stated as follows. Suppose that k is schematic [8].

Theorem 6.5 (correctness of selective CPS transformation)**.**
(1) *If $\Gamma \vdash {e_1}^{\mathsf{p}} : \tau_1 \,@\mathrm{cps}[\tau_2, \tau_2, \mathsf{p}]$ and ${e_1}^{\mathsf{p}} \rightsquigarrow {e_2}^{\mathsf{p}}$, then $\llbracket {e_1}^{\mathsf{p}} \rrbracket_{\mathsf{p}} \sim \llbracket {e_2}^{\mathsf{p}} \rrbracket_{\mathsf{p}}$.*
(2) *If $\Gamma \vdash {e_1}^{\mathsf{i}} : \tau_1 \,@\mathrm{cps}[\tau_2, \tau_2, \mathsf{i}]$ and ${e_1}^{\mathsf{i}} \rightsquigarrow {e_2}^{\mathsf{p}}$, then $\llbracket {e_1}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \,\overline{@}\, k \sim (\underline{\lambda} v.\, k \,\overline{@}\, v) \,\underline{@}\, \llbracket {e_2}^{\mathsf{p}} \rrbracket_{\mathsf{p}}$.*
(3) *If $\Gamma \vdash {e_1}^{\mathsf{i}} : \tau_1 \,@\mathrm{cps}[\tau_2, \tau_3, \mathsf{i}]$ and ${e_1}^{\mathsf{i}} \rightsquigarrow {e_2}^{\mathsf{i}}$, then $\llbracket {e_1}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \,\overline{@}\, k \sim \llbracket {e_2}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \,\overline{@}\, k$.*

Here, $\sim$ denotes β-equality on underlined terms. CPS transformation by Danvy and Filinski preserves reduction [8]. Namely, conclusion of the theorem is β-reduction rather than β-equality. From the theoretical point of view, it is interesting to pursue whether we can attain the preservation of reduction — our future work. Alternatively, we could stay with β-equality but try to simplify as much as possible, as pursued by Davis, Meehan, and Shivers [11].

The proof of the correctness theorem is by induction on the derivation of ${e_1}^{a_1} \rightsquigarrow {e_2}^{a_2}$. We show two interesting cases. The first case is for:

$$\frac{{e_1}^{\mathsf{i}} \rightsquigarrow {e'_1}^{\mathsf{p}}}{([{e_1}^{\mathsf{i}}] \,@^{\mathsf{p}}\, {e_2}^{\mathsf{p}})^{\mathsf{i}} \rightsquigarrow ([{e'_1}^{\mathsf{p}}] \,@^{\mathsf{p}}\, {e_2}^{\mathsf{p}})^{\mathsf{i}}}$$

$$\begin{array}{lcll}
lhs & = & \llbracket ([{e_1}^{\mathsf{i}}] \,@^{\mathsf{p}}\, {e_2}^{\mathsf{p}})^{\mathsf{i}} \rrbracket_{\mathsf{i}} \,\overline{@}\, k & \\
& = & \llbracket {e_1}^{\mathsf{i}} \rrbracket_{\mathsf{i}} \,\overline{@}\, (\overline{\lambda} v_1.\, (\underline{\lambda} v.\, k \,\overline{@}\, v) \,\underline{@}\, (v_1 \,\underline{@}\, \llbracket {e_2}^{\mathsf{p}} \rrbracket_{\mathsf{p}})) & \\
& \sim & (\underline{\lambda} v_1.\, (\underline{\lambda} v.\, k \,\overline{@}\, v) \,\underline{@}\, (v_1 \,\underline{@}\, \llbracket {e_2}^{\mathsf{p}} \rrbracket_{\mathsf{p}})) \,\underline{@}\, \llbracket {e'_1}^{\mathsf{p}} \rrbracket_{\mathsf{p}} & \mathrm{IH} \\
& \sim & (\underline{\lambda} v.\, k \,\overline{@}\, v) \,\underline{@}\, (\llbracket {e'_1}^{\mathsf{p}} \rrbracket_{\mathsf{p}} \,\underline{@}\, \llbracket {e_2}^{\mathsf{p}} \rrbracket_{\mathsf{p}}) & \beta_\Omega \\
& = & \llbracket ([{e'_1}^{\mathsf{p}}] \,@^{\mathsf{p}}\, {e_2}^{\mathsf{p}})^{\mathsf{i}} \rrbracket_{\mathsf{i}} \,\overline{@}\, k & \\
& = & rhs &
\end{array}$$

From the third line to fourth line, we substitute $\llbracket {e'_1}^{\mathsf{p}} \rrbracket_{\mathsf{p}}$ into v_1 (using β_Ω [15]), which preserves β-equality since the first expression to be evaluated is $\llbracket {e'_1}^{\mathsf{p}} \rrbracket_{\mathsf{p}}$ in both lines. However, the third line does not reduce to fourth line if e'_1 is an application. This is where we lose the preservation of β-reduction.

The second case is for shift:

$$\langle F[(\mathcal{S}^{\mathsf{p}} k.\, {e_1}^{\mathsf{p}})^{\mathsf{i}}] \rangle^{\mathsf{p}} \rightsquigarrow \langle (\mathsf{let}\, k = (\lambda^{\mathsf{p}} x.\, \langle F[x^{\mathsf{p}}] \rangle^{\mathsf{p}})^{\mathsf{p}} \;\mathsf{in}\; {e_1}^{\mathsf{p}})^{\mathsf{p}} \rangle^{\mathsf{p}}$$

$$\begin{array}{lcl}
lhs & = & \llbracket \langle F[(\mathcal{S}^{\mathsf{p}} k.\, {e_1}^{\mathsf{p}})^{\mathsf{i}}] \rangle^{\mathsf{p}} \rrbracket_{\mathsf{p}} \\
& = & \llbracket (F[(\mathcal{S}^{\mathsf{p}} k.\, {e_1}^{\mathsf{p}})^{\mathsf{i}}])^{\mathsf{i}} \rrbracket_{\mathsf{i}} \,\overline{@}\, (\overline{\lambda} v.\, v) \\
& \sim & \llbracket (\mathcal{S}^{\mathsf{p}} k.\, {e_1}^{\mathsf{p}})^{\mathsf{i}} \rrbracket_{\mathsf{i}} \,\overline{@}\, (\overline{\lambda} x.\, \llbracket F[x^{\mathsf{p}}] \rrbracket_{\mathsf{i}} \,\overline{@}\, (\overline{\lambda} v.\, v)) \\
& = & \underline{\mathsf{let}}\, k = \underline{\lambda} x.\, \llbracket F[x^{\mathsf{p}}] \rrbracket_{\mathsf{i}} \,\overline{@}\, (\overline{\lambda} v.\, v) \;\underline{\mathsf{in}}\; \llbracket {e_1}^{\mathsf{p}} \rrbracket_{\mathsf{p}} \\
& =^* & \llbracket \langle (\mathsf{let}\, k = (\lambda^{\mathsf{p}} x.\, \langle F[x^{\mathsf{p}}] \rangle^{\mathsf{p}})^{\mathsf{p}} \;\mathsf{in}\; {e_1}^{\mathsf{p}})^{\mathsf{p}} \rangle^{\mathsf{p}} \rrbracket_{\mathsf{p}} \\
& = & rhs
\end{array}$$

From the second line to the third line, we use the following lemma that captures the behavior of shift. Define $\llbracket x^{\mathsf{p}} \rrbracket_{\mathsf{i}}$ to be $\overline{\lambda} k.\, k \,\overline{@}\, x$.

Lemma 6.6. $[\![(F[(S^{a_2}k.\,{e_1}^{a_1})^{\mathsf{i}}])^{\mathsf{i}}]\!]_{\mathsf{i}}\;\overline{@}\;k \sim$
$[\![(S^{a_2}k.\,{e_1}^{a_1})^{\mathsf{i}}]\!]_{\mathsf{i}}\;\overline{@}\;(\overline{\lambda}x.\,[\![F[x^{\mathsf{p}}]]\!]_{\mathsf{i}}\;\overline{@}\;k)$

Proof. By induction on F. □

7 Experiments

We implemented the type inferencer and selective CPS transformer presented in this paper in OCaml. To execute examples, we have extended the source language to include standard constructs, such as pairs, lists, monomorphic let expressions, and sequential execution. All the experiments were performed on MacOSX 10.12.6 with 2.9 GHz Intel Core i5 and 16GB memory. The OCaml version is 4.04.0.

In the examples, we use OchaCaml syntax: $Sk.\,e$ is written as `shift (fun k -> e)` and $\langle e\rangle$ as `reset (fun () -> e)`.

7.1 Prefix

The first example is `prefix`, which produces a list of prefixes of a given list. For example, `prefix [1; 2; 3]` produces `[[1]; [1; 2]; [1; 2; 3]]`.

```
let rec visit lst = match lst with
    [] -> shift (fun k -> [])
  | a :: rest ->
    a :: shift (fun k ->
      k [] :: reset (fun () -> k (visit rest)))
let prefix lst = reset (fun () -> visit lst)
```

The function `prefix` calls `visit` in an empty context. The function `visit` traverses a given list. Whenever a new element is found, it captures the current continuation that conses elements seen so far. Applying it to an empty list constructs the current prefix, while the recursive call produces the rest of the prefixes.

When the above program is fully transformed into CPS, we obtain the following program:[8]

```
let rec visit lst c = match lst with
    [] -> let k v k' = k' (c v) in []
  | a :: rest ->
    let k v k' = k' (c (a :: v)) in
    k [] (fun x ->
    x :: visit rest (fun x -> k x id))
let prefix lst k = k (visit lst id)
```

If we use the selective CPS transformation, we obtain:

```
let rec visit lst c = match lst with
    [] -> let k v = c v in []
  | a :: rest ->
    let k v = c (a :: v) in
    k [] :: visit rest (fun x -> k x)
let prefix lst = visit lst id
```

There are two differences. First, the captured continuations remain in direct style and do not receive the continuation argument. Consequently, application of the captured continuation is also in direct style. Note that it is difficult to handle captured continuations as pure in the standard CPS transformation, because we need to identify all the places where the continuations are used. With selective CPS transformation, captured continuations are naturally kept in direct style (unless they are placed in an impure context). Secondly, the `prefix` function is identified as pure.

We evaluated the above three programs in both OchaCaml and OCaml bytecode. We took prefixes of a list of 3000 elements. We ran the programs 10 times and took the average time. The result is summarized in Table 1. Compared to the standard CPS version, the running time of the selective CPS version reduced to 72% in OchaCaml and 81% in OCaml. This experiment shows that if continuations are heavily captured, we can expect certain amount of speedup by solely transforming captured continuations into direct style.

We also listed running time of the direct-style version in OchaCaml for reference. However, it is not directly comparable to other versions, since their implementation strategies are different. The direct-style version copies the stack whenever continuations are captured, while the other two represent continuations as explicit arguments.

[8] Although we call the resulting program to be in CPS, it contains non-tail calls and thus is not exactly in CPS. This is because the original program contains shift/reset.

Table 1. Running time of `prefix` (in msec). (d) is missing since shift and reset are not supported in OCaml.

			(ms)	ratio
OchaCaml	direct style	(a)	428	
	CPS	(b)	492	
	Selective CPS	(c)	352	0.72 (c/b)
OCaml	direct style	(d)	-	
	CPS	(e)	351	
	Selective CPS	(f)	285	0.81 (f/e)

Table 2. Running time of `queen 11` (in msec).

			(ms)	ratio
OchaCaml	direct style	(a)	873	
	CPS	(b)	1152	
	Selective CPS	(c)	757	0.66 (c/b)
Multicore OCaml	effects	(d)	449	
	CPS	(e)	626	
	Selective CPS	(f)	398	0.64 (f/e)

7.2 *n*-Queen

The second example is the *n*-queen problem.

```
let rec choice num =
  if num = 1 then 1
  else shift (fun k -> k num; k (choice (num-1)))
```

The function `choice` returns a number from 1 to `num` nondeterministically. After capturing the current continuation, it applies the continuation to each number from num down to 1.

```
let queen n =
  let rec loop (i, sol) =
    if i = 0 then print_solution sol
    else let j = choice n in
         let sol2 = j :: sol in
         if is_safe sol2
         then loop (i - 1, sol2)
         else ()
  in loop (n, [])
```

Using choice, the main function queen non-deterministically choose a value j, checks if it is safe to place a queen at j (using is_safe, omitted), and if so, continue. Finally, if we successfully reached to the end, the solution is printed (using print_solution, omitted). The function loop is uncurried, because CPS transformation of curried function results in creation of CPS function for each argument, which would incur too much overhead. For selective CPS transformation, this problem does not arise because passing curried arguments is pure except for the last one.

After selective CPS transformation, choice becomes:

```
let rec choice num c =
  if num = 1 then c 1
  else let k v = c v in
       (k num; choice (num - 1) (fun x -> k x))
```

As before, the captured continuation is in direct style. In the CPS version (omitted), k is transformed into CPS. Here is the queen function:

```
let queen n =
  let rec loop (i, sol) c =
    if i = 0 then c (print_solution sol)
    else choice n (fun j ->
         let sol2 = j :: sol in
         if is_safe sol2
         then loop (i - 1, sol2) (fun x -> c x)
         else c ())
  in loop (n, []) id
```

We observe that is_safe, print_solution, and queen itself are in direct style. In the CPS version (omitted), they are all transformed into CPS, including the bodies of is_safe and print_solution. It has significant impact on performance.

For this experiment, we used OCaml 4.02.2 with multicore support (Multicore OCaml [12]) instead of the standard OCaml. Multicore OCaml supports effect handlers [24] that can capture the current continuation similarly to shift. When the captured continuation is used multiple times (e.g., for backtracking), stack copying is used. For reference, we have implemented n-queen problem in direct style using the effect handlers in Multicore OCaml, too.

Table 2 summarizes the result of running 11-queen problem. (Numbers other than 11 had similar results.) The figures are the average of ten runs. In both OchaCaml and Multicore OCaml, the running time is reduced to around 66%, thanks to the direct-style execution of captured continuations and is_safe, both being in a loop and hence executed many times. Again, we cannot directly compare the running time of the direct-style version with other versions, since their implementation strategies are different. However, in all cases, we observe the direct-style versions perform better than the CPS version, but worse than the selective CPS version.

8 Library Functions

Realistic languages have many library functions. In a language with shift/reset that supports answer type modification, we cannot directly link existing library functions, because they are typed with the traditional type system without answer types. If we embed a program with shift/reset into a standard functional language via selective CPS transformation, we are able to use existing library functions. However, since impure functions are transformed into CPS, a problem arises for higher-order library functions.

To use higher-order library functions, we need to annotate them as completely pure. For example, List.map in OCaml is given the following type:

$$(\alpha \to \beta @\mathrm{cps}[\gamma_1, \gamma_1, \mathrm{p}]) \to$$
$$(\alpha\ \mathrm{list} \to \beta\ \mathrm{list} @\mathrm{cps}[\gamma_2, \gamma_2, \mathrm{p}]) @\mathrm{cps}[\gamma_3, \gamma_3, \mathrm{p}]$$

where the first argument is explicitly constrained to a pure function. When an impure function is given as a first argument, a type error occurs (the "type error" case in Figure 7). To allow an impure function in such a case, one could prepare for another version of List.map where the first argument is classified as impure, as is done in Koka [20].

9 Related Work

Kim, Yi, and Danvy [16] use selective CPS transformation to compile away exception handling and report that exception-intensive programs can benefit from the selective CPS transformation. Nielsen [23] presents a (two-pass) selective CPS transformation for a calculus with call-with-current-continuation and throw and shows its correctness via colon translation. Our work is a non-trivial extension of their work to shift/reset where answer types play an important role.

Materzok and Biernacki [22] define a type system for control operators, shift_0 and reset_0, and compile them into simply-typed λ-calculus via selective CPS transformation. They show a type inference algorithm that finds the principal type, but with unsolved constraints. We show that the constraint solving is non-trivial for shift and reset case: some terms do not have the best annotation.

Apart from supporting control operators in a host language without them, Danvy and Hatcliff [9] identify strict parts of a program in a lazy language and transform the program into CPS by applying call-by-value or call-by-name CPS transformation selectively. Danvy and Hatcliff [10] also show how to transform a direct-style interpreter into a natural CPS interpreter by identifying possibly non-terminating parts and transforming those parts selectively. Reppy [25] transforms designated function calls selectively into CPS to improve efficiency of nested loops.

10 Conclusion

In this paper, we have presented a one-pass selective CPS transformation that compiles away control operators, shift and reset, into a standard functional language and proved its correctness. We have not been able to attain preservation of β-reductions, only preservation of β-equality. It is our future work to investigate if it is possible to obtain preservation of β-reduction by reformulating the transformation. We conjecture that we need pure terms to be translated into A-normal form. Another future work would be formalization using a proof assistant. The selective CPS transformation dispatches over purity annotations of subterms and has a lot of cases. It is becoming unrealistic to check all the cases manually. We have already started formalization in Agda and it spotted some errors in the manual proof.

Acknowledgments

We are grateful to Youyou Cong, Yukiyoshi Kameyama, and anonymous reviewers for many comments and discussions. This work was partly supported by JSPS KAKENHI under Grant No. JP15K00090.

References

[1] Asai, K. "On Typing Delimited Continuations: Three New Solutions to the Printf Problem," *Higher-Order and Symbolic Computation*, Vol. 22, No. 3, pp. 275–291, Kluwer Academic Publishers (September 2009).

[2] Asai, K., and Y. Kameyama "Polymorphic Delimited Continuations," *Proceedings of the Fifth Asian Symposium on Programming Languages and Systems, LNCS 4807*, pp. 239–254 (November 2007).

[3] Asai, K., and O. Kiselyov "Introduction to Programming with Shift and Reset," Tutorial notes on delimited continuations at Continuation Workshop 2011, available from `http://pllab.is.ocha.ac.jp/~asai/cw2011tutorial/main-e.pdf`, 14 pages (September 2011).

[4] Biernacka, M., D. Biernacki, and O. Danvy "An Operational Foundation for Delimited Continuations in the CPS Hierarchy," *Logical Methods in Computer Science*, Vol. 1, No. 2:5, pp. 1–39 (November 2005).

[5] Danvy, O. "Functional Unparsing," *Journal of Functional Programming*, Vol. 8, No. 6, pp. 621–625, Cambridge University Press (November 1998).

[6] Danvy, O., and A. Filinski "A Functional Abstraction of Typed Contexts," Technical Report 89/12, DIKU, University of Copenhagen (July 1989).

[7] Danvy, O., and A. Filinski "Abstracting Control," *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pp. 151–160 (June 1990).

[8] Danvy, O., and A. Filinski "Representing Control, a Study of the CPS Transformation," *Mathematical Structures in Computer Science*, Vol. 2, No. 4, pp. 361–391 (December 1992).

[9] Danvy, O., and J. Hatcliff "CPS-Transformation After Strictness Analysis," *ACM Letters on Programming Languages and Systems*, Vol. 1, No. 3, pp. 195–212 (September 1992).

[10] Danvy, O., and J. Hatcliff "On the transformation between direct and continuation semantics," In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt editors, *Mathematical Foundations of Programming Semantics (LNCS 802)*, pp. 627–648 (April 1993).

[11] Davis, M., W. Meehan, and O. Shivers "No-Brainer CPS Conversion (Functional Pearl)," *Proceedings of the ACM on Programming Languages*, Vol. 1, Issue ICFP, Article No. 23, 25 pages (September 2017).

[12] Dolan, S., L. White, and A. Madhavapeddy "Multicore OCaml," OCaml Workshop, 2 pages (May 2014).

[13] Filinski, A. "Representing Monads," *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages*, pp. 446–457 (January 1994).

[14] Henglein, F. "Efficient Type Inference for Higher-Order Binding-Time Analysis," In J. Hughes, editor, *Functional Programming Languages and Computer Architecture (LNCS 523)*, pp. 448–472 (August 1991).

[15] Kameyama, Y., and M. Hasegawa "A Sound and Complete Axiomatization of Delimited Continuations," *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming*, pp. 177–188 (August 2003).

[16] Kim, J., K. Yi, and O. Danvy "Assessing the Overhead of ML Exceptions by Selective CPS Transformation," *Proceedings of the 1998 ACM SIGPLAN Workshop on ML*, pp.103–114 (September 1998). Also in BRICS Research Report RS-98-15, Department of Computer Science, Aarhus University, 34 pages (September 1998).

[17] Kiselyov, O. "Delimited Control in OCaml, Abstractly and Concretely: System Description," In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming (LNCS 6009)*, pp. 304–320 (April 2010).

[18] Kobori, I., Y. Kameyama, and O. Kiselyov "Answer-Type Modification without Tears: Prompt-Passing Style Translation for Typed Delimited-Control Operators," *Workshop on Continuations 2015, Electronic Proceedings in Theoretical Computer Science* 212, pp. 36–52 (June 2016).

[19] Lawall, J. L., and O. Danvy "Continuation-Based Partial Evaluation," *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pp. 227–238 (June 1994).

[20] Leijen, D. "Type Directed Compilation of Row-Typed Algebraic Effects," *Proceedings of the 44rd ACM SIGPLAN Symposium on Principles of Programming Languages*, pp. 486–499 (January 2017).

[21] Masuko, M., and K. Asai "Caml Light + shift/reset = Caml Shift," *Theory and Practice of Delimited Continuations*, pp. 33–46 (May 2011).

[22] Materzok, M., D. Biernacki "Subtyping Delimited Continuations," *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, pp. 81–93 (September 2011).

[23] Nielsen, L. R. "A Selective CPS Transformation," *Electronic Notes in Theoretical Computer Science* Vol. 45, pp. 311–331 (November 2001).

[24] Pretnar, M. "An Introduction to Algebraic Effects and Handlers, Invited Tutorial Paper," *Mathematical Foundations of Programming Semantics, Electronic Notes in Theoretical Computer Science* 319, pp. 19–35 (June 2015)

[25] Reppy, J. "Local CPS Conversion in a Direct-style Compiler," *Proceedings of the third ACM SIGPLAN Workshop on Continuations*, pp. 13–22 (January 2001).

[26] Rompf, T., I. Maier, and M. Odersky "Implementing First-Class Polymorphic Delimited Continuations by a Type-Directed Selective CPS-Transform," *Proceedings of the 2009 ACM SIGPLAN International Conference on Functional Programming*, pp. 317–328 (August 2009).

[27] Thielecke, H. "From Control Effects to Typed Continuation Passing," *Conference Record of the 30th Annual ACM Symposium on Principles of Programming Languages*, pp. 139–149 (January 2003).

[28] Thiemann, P. J. "Cogen in Six Lines," *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pp. 180–189 (May 1996).

[29] Wright, A. K., and M. Felleisen "A Syntactic Approach to Type Soundness," *Information and Computation*, Vol. 115, No. 1, pp. 38–94 (November 1994).