Electronic Notes in Theoretical Computer Science 45 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume45.html 21 pages

A Selective CPS Transformation

Lasse R. Nielsen

BRICS[1]

Department of Computer Science, University of Aarhus
Building 540, Ny Munkegade, DK-8000 Aarhus C, Denmark.
E-mail: lrn@brics.dk

Abstract

The CPS transformation makes all functions continuation-passing, uniformly. Not all functions, however, need continuations: they only do if their evaluation includes computational effects. In this paper we focus on control operations, in particular "call with current continuation" and "throw". We characterize this involvement as a control effect and we present a selective CPS transformation that makes functions and expressions continuation-passing if they have a control effect, and that leaves the rest of the program in direct style. We formalize this selective CPS transformation with an operational semantics and a simulation theorem à la Plotkin.

1 Introduction

This paper defines, and proves correct, a selective Continuation-Passing Style (CPS) transformation, i.e., one that preserves part of the program in direct style. It uses information about a program's effect-behavior to guide the transformation. The particular computational effect we use is the control-transfer effect exemplified by "call with current continuation" [3].

1.1 Related work

Selectively CPS transforming a program is not a new idea.

Danvy and Hatcliff defined a selective CPS transformation based on strictness analysis [6]. When dealing with the effect of non-termination, a strict function is indifferent to the evaluation order of its argument, and as such arguments of strict functions could be transformed by a call-by-value transformation while the rest of the program was transformed by a call-by-name transformation. The same authors also investigated CPS transformation based

[1] Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

on totality information, defining a selective transformation [7]. There is no immediate generalization of strictness to other effects than non-termination, though, whereas triviality (the absence of effects) generalizes immediately to all other computational effects, as we have exemplified with control effects.

Kim, Yi, and Danvy implemented a selective CPS transformation for the core language of SML of New Jersey to reduce the overhead of their CPS-transformation which λ-encoded the exception effects of SML [15]. The implementation is very similar to the present work, though they treat exceptions instead of control operations and base the annotation on a specific exception analysis.

Recent work has focused on selective transformations for different reasons.

Reppy introduced a local CPS transformation in an otherwise direct-style compiler to improve the efficiency of nested loops [18]. Kim and Yi defined coercions between direct style and CPS terms with no other effects than non-termination, allowing arbitrary subexpressions to be transformed, and facilitating interfacing to external code in both direct style and CPS. The selective transformation was proven correct [14].

The present paper defines a selective CPS transformation for the simply typed λ-calculus extended with recursive functions and computational effects, namely CALLCC and THROW, into λ-calculus with only recursive functions, i.e., with only non-termination as an effect. The transformation is based on an effect analysis, and it is proven correct with respect to the dynamic behavior of the program. We conjecture that the methodology extends to other types of effects, only differing in the details of the encoding of effectful primitives into λ-expressions.

1.2 Overview

Section 2 gives the syntax and semantics of a small, typed functional language with control effects, and shows the traditional (non-selective) CPS transformation. Section 3 extends the language with effect annotations which are verified by an effect system, and defines the selective CPS transformation guided by these annotations. Section 4 proves the correctness of the selective CPS transformation using Plotkin-inspired colon translations, and Section 5 concludes.

2 Definitions

We define the source and target language of our selective CPS transformation, giving the syntax and type predicates as well as an operational semantics.

2.1 Syntax

The source language is a call-by-value typed functional language with recursive functions and the canonical control operators: CALLCC and THROW. The

syntax is given in Figure 1, where x and f range over a set of identifiers and c ranges over a set of constants.

$$e \ ::= \ \mathrm{c} \mid \mathrm{x} \mid \textsc{fun}\ \mathrm{f}\ \mathrm{x}.e \mid e \ @ \ e \mid \textsc{callcc}\ \mathrm{x}.e \mid \textsc{throw}\ e\ e$$

Fig. 1. Abstract syntax of the source language

We identify expressions up to renaming of bound variables, i.e., FUN f x.x = FUN g y.y. We use the shorthand $\lambda \mathrm{x}.e$ for a function-abstraction FUN f x.e where f does not occur in e.

We require expressions to be well typed with regard to the typing rules in Figure 2, similar to those given by Harper, Duba, and MacQueen [12].

The syntax of types is given by the grammar:

$$\tau ::= b \mid \tau \rightarrow \tau \mid \langle \tau \rangle$$

where b ranges over a set of base types, and $\langle \tau \rangle$ is the type of continuations expecting a value of type τ.

In the rules ConstType is a mapping from constants to base types, giving the type of a constant, and Γ is a mapping from identifiers to types.

$$\frac{\Gamma(\mathrm{x}) = \tau}{\Gamma \vdash \mathrm{x} : \tau} \qquad \frac{\textsc{ConstType}(\mathrm{c}) = b}{\Gamma \vdash \mathrm{c} : b}$$

$$\frac{\Gamma; \mathrm{x} : \tau_1; \mathrm{f} : \tau_1 \rightarrow \tau_2 \vdash e : \tau_2}{\Gamma \vdash \textsc{fun}\ \mathrm{f}\ \mathrm{x}.e : \tau_1 \rightarrow \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 \ @ \ e_2 : \tau_2}$$

$$\frac{\Gamma; \mathrm{x} : \langle \tau \rangle \vdash e : \tau}{\Gamma \vdash \textsc{callcc}\ \mathrm{x}.e : \tau} \qquad \frac{\Gamma \vdash e_1 : \langle \tau \rangle \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \textsc{throw}\ e_1\ e_2 : \tau_2}$$

Fig. 2. The typing rules of the source language.

We define a program to be a closed expression of type b_0, an arbitrary base type, and we only prove the correctness of the transformation on entire programs.

2.2 Semantics

The source language has a left-to-right call-by-value(CBV) operational semantics given using evaluation contexts. We introduce an intermediate value

to the syntax to represent captured continuations, and define the values and contexts recursively, and give the context-based reduction rules (Figure 3).

Extended expressions, values, and evaluation contexts:

$$e ::= \ldots \mid \langle \mathrm{E} \rangle$$
$$v ::= \mathrm{c} \mid \textsc{fun}\ \mathrm{f}\ \mathrm{x}.e \mid \langle \mathrm{E} \rangle$$
$$\mathrm{E} ::= [\,] \mid \mathrm{E}\ @\ e \mid v\ @\ \mathrm{E} \mid \textsc{throw}\ \mathrm{E}\ e \mid \textsc{throw}\ v\ \mathrm{E}$$

Reduction rules:

$$\mathrm{E}[(\textsc{fun}\ \mathrm{f}\ \mathrm{x}.e)\ @\ v] \rightarrow \mathrm{E}[e\,[\textsc{fun}\ \mathrm{f}\ \mathrm{x}.e/\mathrm{f}]\,[v/\mathrm{x}]]$$

$$\mathrm{E}[\textsc{callcc}\ \mathrm{x}.e] \rightarrow \mathrm{E}[e\,[\langle \mathrm{E} \rangle/\mathrm{x}]]$$

$$\mathrm{E}[\textsc{throw}\ \langle \mathrm{E}_1 \rangle\ \ v] \rightarrow \mathrm{E}_1[v]$$

Fig. 3. Semantics of the source language

Notice that continuations are represented as contexts, so throwing a continuation amounts to reinstating a context.

This semantics uses evaluation contexts in a way similar to Felleisen [9], capturing the fact that for any expression, there is at most one enabled reduction at a time.

Subject reduction ($e : \tau$ and $e \rightarrow e'$ implies $e' : \tau$ where the rules have been extended to include contexts), can be proven by a completely standard proof, which has been omitted.

2.3 The CPS transformation

The CPS transformation can be used to transform a program in the source language into a program in a similar language without CALLCC and THROW, while preserving the computational behavior of the source program. That is, the translated program, when applied to an initial continuation, terminates if the source program did, and the result of the transformed program, which is of a base type, is the same as that of the original program.

The CPS-transformation has other interesting properties:

- It generates programs that can be evaluated under both a call-by-name (CBN) and a CBV semantics and yield the same result, which corresponds to the result of the source program. A number of CPS transformations exist, each corresponding to an evaluation-order for the source language [13].

- It generates programs where all applications are tail-calls, i.e., no application is in an argument position.

CPS transformations are used in many places, but primarily in compilers to give an intermediate representation [1,21] and as a way to simplify the language of a program before applying other transformations or analyzes to it [4]. It is the last application that is the motivation for the present work.

The standard CBV CPS transformation is defined in Figure 4.

$$
\begin{aligned}
\mathcal{C}[\![\mathrm{c}]\!] &= \lambda \mathrm{k}.\mathrm{k} \mathbin{@} \mathcal{C}_t[\![\mathrm{c}]\!] \\
\mathcal{C}[\![\mathrm{x}]\!] &= \lambda \mathrm{k}.\mathrm{k} \mathbin{@} \mathcal{C}_t[\![\mathrm{x}]\!] \\
\mathcal{C}[\![\textsc{fun}\ \mathrm{f}\ \mathrm{x}.e]\!] &= \lambda \mathrm{k}.\mathrm{k} \mathbin{@} \mathcal{C}_t[\![\textsc{fun}\ \mathrm{f}\ \mathrm{x}.e]\!] \\
\mathcal{C}[\![e_1 \mathbin{@} e_2]\!] &= \lambda \mathrm{k}.\mathcal{C}[\![e_1]\!] \mathbin{@} \lambda \mathrm{v}.\mathcal{C}[\![e_2]\!] \mathbin{@} \lambda \mathrm{v}'.\mathrm{v} \mathbin{@} \mathrm{v}' \mathbin{@} \mathrm{k} \\
\mathcal{C}[\![\textsc{callcc}\ \mathrm{x}.e]\!] &= \lambda \mathrm{k}.\lambda \mathrm{x}.\mathcal{C}[\![e]\!] \mathbin{@} \mathrm{k} \mathbin{@} \mathrm{k} \\
\mathcal{C}[\![\textsc{throw}\ e_1\ e_2]\!] &= \lambda \mathrm{k}.\mathcal{C}[\![e_1]\!] \mathbin{@} \lambda \mathrm{v}.\mathcal{C}[\![e_2]\!] \mathbin{@} \mathrm{v}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}_t[\![\mathrm{x}]\!] &= \mathrm{x} \\
\mathcal{C}_t[\![\mathrm{c}]\!] &= \mathrm{c} \\
\mathcal{C}_t[\![\textsc{fun}\ \mathrm{f}\ \mathrm{x}.e]\!] &= \textsc{fun}\ \mathrm{f}\ \mathrm{x}.\mathcal{C}[\![e]\!]
\end{aligned}
$$

Fig. 4. The CPS transformation

The $\mathcal{C}_t[\![\cdot]\!]$ function is used to coerce values and identifiers into CPS form, which consists of transforming the bodies of function abstractions. Values and identifiers share the property that Reynolds called "being trivial" [19] and Moggi called "being a value" (as opposed to a computation) [16], in the sense that they have no computational effects, including nontermination. The $\mathcal{C}[\![\cdot]\!]$ function is used on expressions with potential effects, the "serious" expressions.

3 The selective CPS transformation

As stated in the previous section, we treat trivial and serious expressions differently. Trivial expressions are those that have no computational effects and serious expressions are those that might have effects. The safe approximation used by the standard CPS transformation assumes that any application might have effects, which is not unreasonable when one considers nontermination as an effect.

In the source language we have added control effects, and it makes sense only to focus on those, and let the termination behavior be preserved by only evaluating the result in a CBV semantics. If we do so, we can use an effect analysis to find the parts of the program that are guaranteed to be free of the control effects generated by CALLCC and THROW. In the following we will use the words "trivial" and "non-trivial" about the absence or possible presence of *control* effects only, while ignoring the partiality effect of non-termination. That is, an expression that has an infinite reduction sequence can still be said to be "trivial" with regards to control effects. We are not aiming for evaluation-order independence, rather the source and target languages are assumed to have the same evaluation-order (call-by-value), so the translation need not take any measures to preserve or prevent non-termination in otherwise effect-free expressions.

This section defines effect-annotated expressions, an effect type system to check the consistency of the annotation, and a selective CPS transformation that keeps trivial applications in direct style.

3.1 Annotated source language

We annotate a program with annotations taken from the set $\{\mathrm{T}, \mathrm{N}\}$, which is a partial order with the ordering relation $\mathrm{N} < \mathrm{T}$.

We mark some applications as trivial, with a T, and some as (potentially) non-trivial, with an N. The trivial ones are kept in direct style, and as such do not expect to receive a continuation.

For an expression, an effect analysis can tell us one of three things:

- Some evaluation of the expression will certainly give rise to effects (meaning, in this case, the reduction sequence of the expression contains evaluations of CALLCC or THROW expressions),
- no evaluation of the expression will give rise to effects, or
- we just don't know either way, which can happen since the problem is generally undecidable.

These three options corresponds to a three pointed domain, where both certainty of effects and certainty of absence of effects are above the uncertainty. The present transformation aims to keep the expressions in the second case in direct style. Since any effectful expression *must* be put into CPS, we must treat the first and third cases equally, and they are both marked N. Unifying these two cases gives rise to the stated ordering where greater means more information is known: certainty of the absence of effects as opposed to only possible presence, only losing information when the two cases are unified.

$$\mathrm{A} ::= \mathrm{T} \mid \mathrm{N}$$

$$e ::= \mathrm{c} \mid \mathrm{x} \mid \textsc{fun}^{\mathrm{A}}\ \mathrm{f}\ \mathrm{x}.e \mid (e\ @^{\mathrm{A}}\ e)^{\mathrm{A}} \mid \textsc{callcc}\ \mathrm{x}.e \mid \textsc{throw}\ e\ e$$

These are the minimal annotations needed for our purpose. We treat values and identifiers (the traditional trivial expressions) as if they were annotated as such, i.e., $(e)^{\mathrm{T}}$ is a match for any trivial expression, just as $(e)^{\mathrm{N}}$ matches the two control operators.

We require that an expression annotated as trivial actually is so, i.e., whenever it is evaluated, the reduction sequence contains no steps corresponding to reductions of CALLCC or THROW expressions. Since we use this annotation as a basis for the selective CPS transformation, we will want to CPS transform all expressions that are not marked trivial.

We have to treat functions and applications with special care. When a lambda abstraction is applied at an application point, the body of the abstraction is also evaluated at that point. If the body is not trivial, then neither is the application, and after selective CPS transformation, the transformed application must pass a continuation to the transformed body, and the body should expect a continuation.

In a higher-order program, more than one abstraction can be applied at the same application point, and after transformation, all of these abstractions must either expect a continuation or not. That means that all functions that can end up in a given application must be transformed in the same way. That divides the abstractions into two groups, those transformed into CPS, i.e., expecting a continuation, and those kept in direct style, i.e., not expecting a continuation. Some abstractions with a trivial body might be transformed to expect a continuation in order to match the other abstractions that reach the same application points.

We will say that the annotation is "consistent" (with regards to the behavior of the program) if:

- All expressions marked trivial are trivial,
- all abstractions marked trivial, or non-trivial, are only applied at application points marked trivial, or non-trivial respectively, and
- all abstractions whose body are marked non-trivial, are themselves marked as non-trivial.

To check all this, we use an extension of the type system to an effect type system that guarantees that the annotation is consistent. The types are also annotated, so the grammar of types is:

$$\tau ::= b \mid \tau \xrightarrow{\mathrm{A}} \tau \mid \langle \tau \rangle$$

The effect system is shown in Figure 5.

If an expression is typeable in the original type system, then there exists at least one annotation that is typeable in the effect system, namely the one where all functions and applications are marked non-trivial.

The $\leq$ in the rule for function abstractions is exactly due to the restriction on which functions can flow where. Functions with trivial bodies can be

$$\frac{\Gamma(\mathrm{x}) = \tau}{\Gamma \vdash \mathrm{x} : \tau, \mathrm{T}} \qquad \frac{\textsc{ConstType}(\mathrm{c}) = b}{\Gamma \vdash \mathrm{c} : b, \mathrm{T}}$$

$$\frac{\Gamma[\mathrm{x} : \tau_1][\mathrm{f} : (\tau_1 \xrightarrow{\mathrm{A}_1} \tau_2)] \vdash e : \tau_2, \mathrm{A} \quad \mathrm{A}_1 \leq \mathrm{A}}{\Gamma \vdash \textsc{fun}\ \mathrm{f}\ \mathrm{x}.e : \tau_1 \xrightarrow{\mathrm{A}_1} \tau_2, \mathrm{T}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{\mathrm{A}_3} \tau_2, \mathrm{A}_1 \quad \Gamma \vdash e_2 : \tau_1, \mathrm{A}_2 \quad \mathrm{A} \leq \min(\mathrm{A}_1, \mathrm{A}_2, \mathrm{A}_3)}{\Gamma \vdash e_1\ @\ e_2 : \tau_2, \mathrm{A}}$$

$$\frac{\Gamma[\mathrm{x} : \langle \tau \rangle] \vdash e : \tau, \mathrm{A}}{\Gamma \vdash \textsc{callcc}\ \mathrm{x}.e : \tau, \mathrm{N}} \qquad \frac{\Gamma \vdash e_1 : \langle \tau \rangle, \mathrm{A}_1 \quad \Gamma \vdash e_2 : \tau, \mathrm{A}_2}{\Gamma \vdash \textsc{throw}\ e_1\ e_2 : \tau_2, \mathrm{N}}$$

Fig. 5. Effect type system

annotated with an N, allowing them to flow into an application expecting to pass a continuation, but this is reflected in the type, which is the only thing that is known at the application point. There are two different annotated function types, one for each annotation, and only one of these is allowed at each application point.

The $\leq$ on the rule for applications is discussed in the next section,

We only consider well-annotated expressions from here on, i.e., expressions that are allowed by the effect typing rules.

3.2 The Selective CPS Transformation

We define a CPS transformation that transforms well-annotated expressions and leaves trivial applications in direct style (Figure 6).

3.3 Semantics of annotated syntax

We do not change the semantics of the language, since the annotation is just a mark on the expressions, and it is only used by the CPS transformation. Still, in order to prove the correctness of the transformation, we define a reduction relation on annotated expressions that updates the annotation as well.

$$\begin{aligned}
\mathrm{E}\big[((\textsc{fun}^{\mathrm{A}_1}\ \mathrm{f}\ \mathrm{x}.(e)^{\mathrm{A}_3})\ @^{\mathrm{A}_2}\ v)^{\mathrm{A}}\big] &\rightarrow \mathrm{E}\big[(e)^{\mathrm{A}_3}\ \big[\textsc{fun}^{\mathrm{A}_1}\ \mathrm{f}\ \mathrm{x}.(e)^{\mathrm{A}_3}/\mathrm{f}\big]\ [v/\mathrm{x}]\big] \\
\mathrm{E}\big[\textsc{callcc}\ \mathrm{x}.(e)^{\mathrm{A}}\big] &\rightarrow \mathrm{E}\big[(e)^{\mathrm{A}}\ [\langle \mathrm{E} \rangle/\mathrm{x}]\big] \\
\mathrm{E}[\textsc{throw}\ \langle \mathrm{E}' \rangle\ v] &\rightarrow \mathrm{E}'[v]
\end{aligned}$$

The point of this reduction relation is that values and identifiers are always marked trivial, and no expression marked trivial can ever reduce to one marked

$$
\begin{aligned}
\mathcal{S}[\![e^{\mathrm{T}}]\!] &= \mathrm{k} \mathbin{@} \mathcal{S}_t[\![e^{\mathrm{T}}]\!] \\
\mathcal{S}[\![(e_1 \mathbin{@^{\mathrm{N}}} e_2)^{\mathrm{N}}]\!] &= \lambda \mathrm{k}.\mathcal{S}[\![e_1]\!] \mathbin{@} (\lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \mathbin{@} (\lambda \mathrm{v}'.\mathrm{v} \mathbin{@} \mathrm{v}' \mathbin{@} \mathrm{k})) \\
\mathcal{S}[\![(e_1 \mathbin{@^{\mathrm{T}}} e_2)^{\mathrm{N}}]\!] &= \lambda \mathrm{k}.\mathcal{S}[\![e_1]\!] \mathbin{@} (\lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \mathbin{@} (\lambda \mathrm{v}'.\mathrm{k} \mathbin{@} (\mathrm{v} \mathbin{@} \mathrm{v}'))) \\
\mathcal{S}[\![\textsc{callcc}\ \mathrm{x}.e]\!] &= \lambda \mathrm{k}.(\lambda \mathrm{x}.\mathcal{S}[\![e]\!] \mathbin{@} \mathrm{k}) \mathbin{@} \mathrm{k} \\
\mathcal{S}[\![\textsc{throw}\ e_1\ e_2]\!] &= \lambda \mathrm{k}.\mathcal{S}[\![e_1]\!] \mathbin{@} (\lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \mathbin{@} \mathrm{v})
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{S}_t[\![\mathrm{x}]\!] &= \mathrm{x} \\
\mathcal{S}_t[\![\mathrm{c}]\!] &= \mathrm{c} \\
\mathcal{S}_t[\![\textsc{fun}^{\mathrm{N}}\ \mathrm{f}\ \mathrm{x}.e]\!] &= \textsc{fun}\ \mathrm{f}\ \mathrm{x}.\mathcal{S}[\![e]\!] \\
\mathcal{S}_t[\![\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e]\!] &= \textsc{fun}\ \mathrm{f}\ \mathrm{x}.\mathcal{S}_t[\![e]\!] \\
\mathcal{S}_t[\![(e_1 \mathbin{@^{\mathrm{T}}} e_2)^{\mathrm{T}}]\!] &= \mathcal{S}_t[\![e_1]\!] \mathbin{@} \mathcal{S}_t[\![e_2]\!]
\end{aligned}
$$

Fig. 6. The selective CPS transformation

as non-trivial.

With these reduction rules, an expression marked non-trivial can reduce to one marked trivial, typically by reducing it to a value. If that happens to one of the subexpressions of an application, we can suddenly be in the situation where both of the subexpressions are trivial as well as the bodies of the functions expected to be applied there, and the entire application could now be consistently annotated as trivial. The weakening in the effect-typing rule for applications is there to avoid that such a change would mandate changes to annotations not local to the reduction taking place.

All these properties make a proof of Subject Reduction a trivial extension of the proof for the unannotated syntax.

One reason for having both annotations and an effect system, and not, e.g., only the effect system, is for ease of representation. Even if a reduced program allows a more precise effect-analysis than the original program, the transformation is based on the original program, and the annotated reduction keeps the original annotation throughout the reduction sequence.

4 Proof of correctness

To prove the correctness of the transformation, we must first specify a notion of correctness. In this case we require that the transformed program reduces to the same result as the original program.

Theorem 4.1 (Correctness of the Selective CPS Transformation) *If e is a closed and well-annotated expression of type b_0 then*

$$e \rightarrow^* v \Leftrightarrow \mathcal{S}[\![e]\!] \mathbin{@} (\lambda \mathrm{x.x}) \rightarrow^* v$$

In Plotkin's original proof, the result of the transformed program would be $\mathcal{S}_t[\![v]\!]$, but since the program has a type where the only values are constants, and all constants satisfy $\mathcal{S}_t[\![\mathrm{c}]\!] = \mathrm{c}$, we can state the theorem as above.

4.1 The selective colon-translations

The proof uses a method similar to Plotkin's in his original proof of the correctness of the CPS transformation [17]. It uses a so-called "colon-translation" to bypass the initial administrative reductions and focus on the evaluation point.

The intuition that drives the normal CPS transformation is that if e reduces to v then $(\mathcal{C}[\![e]\!] \mathbin{@} k)$ should evaluate to $(k \mathbin{@} \mathcal{C}_t[\![v]\!])$. Plotkin captured this in his colon translation where if $e \rightarrow e'$ then $e : k \rightarrow^* e' : \mathrm{k}$, and at the end of the derivation, values satisfied $v : k = k \mathbin{@} \Psi(v)$, where $\Psi(\cdot)$ is what we write $\mathcal{S}_t[\![\cdot]\!]$.

The idea of the colon translation is that in $e : k$, the k represents the *context* of e, which in the transformed program has been collected in a continuation: a function expecting the result of evaluating e. The colon separates the source program to the left and the transformed program to the right of it. In the selective CPS transform, some contexts are not turned into continuations, namely the contexts of expressions marked trivial, since such expressions are not transformed to CPS expressions, and as such does not expect a continuation.

Therefore we have two colon translations, one for non-trivial expressions, with a continuation function after the colon, and one for trivial expressions with an evaluation context after the colon. The definition is shown in Figure 7. In both cases, what is to the left of the colon is a piece of source syntax, and what is to the right is a representation of the context of that expression in the source program translated to the target language. If the expression is trivial, the source context is represented by a context in the target language, and the translation of the expression is put into this context. If the expression is not trivial, then the source context is represented by a continuation function which is passed to the translation of the expression.

In Plotkin's colon translation, $v : k = k \mathbin{@} \Phi(v)$. This also holds for this colon translation pair, since $v : k = v : [k \mathbin{@} [\,]]$, since v is trivial, and $v : [k \mathbin{@} [\,]] = k \mathbin{@} v$ by the definition of the $e : \mathrm{E}$-translation.

The $e : \mathrm{E}$-translation is not as significant as the $e : k$-translation, since all it does is apply the $\mathcal{S}_t$-function to the argument, i.e., if e is a trivial expression then $e : \mathrm{E} = \mathrm{E}[\mathcal{S}_t[\![e]\!]]$. There are no administrative reductions to bypass in direct style.

We plan to use the colon translations on the result of reducing on the

$$
\begin{array}{rcll}
e^{\mathrm{T}} : k & = & e^{\mathrm{T}} : [k \mathbin{@} [\,]] & \\
(e_1 \mathbin{@}^{\mathrm{N}} e_2)^{\mathrm{N}} : k & = & e_1 : \lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \mathbin{@} (\lambda \mathrm{v}'.\mathrm{v} \mathbin{@} \mathrm{v}' \mathbin{@} k) & \text{if } e_1 \text{ is not a value} \\
(v_1 \mathbin{@}^{\mathrm{N}} e_2)^{\mathrm{N}} : k & = & e_2 : \lambda \mathrm{v}'.\mathcal{S}_t[\![v_1]\!] \mathbin{@} \mathrm{v}' \mathbin{@} k & \text{if } e_2 \text{ is not a value} \\
(v_1 \mathbin{@}^{\mathrm{N}} v_2)^{\mathrm{N}} : k & = & \mathcal{S}_t[\![v_1]\!] \mathbin{@} \mathcal{S}_t[\![v_2]\!] \mathbin{@} k & \\
(e_1 \mathbin{@}^{\mathrm{T}} e_2)^{\mathrm{N}} : k & = & e_1 : \lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \mathbin{@} (\lambda \mathrm{v}'.k \mathbin{@} (\mathrm{v} \mathbin{@} \mathrm{v}')) & \text{if } e_1 \text{ is not a value} \\
(v_1 \mathbin{@}^{\mathrm{T}} e_2)^{\mathrm{N}} : k & = & e_2 : \lambda \mathrm{v}'.k \mathbin{@} (\mathcal{S}_t[\![v_1]\!] \mathbin{@} \mathrm{v}') & \text{if } e_2 \text{ is not a value} \\
(v_1 \mathbin{@}^{\mathrm{T}} v_2)^{\mathrm{N}} : k & = & k \mathbin{@} (\mathcal{S}_t[\![v_1]\!] \mathbin{@} \mathcal{S}_t[\![v_2]\!]) & \\
\textsc{callcc}\ \mathrm{x}.e : k & = & (\lambda \mathrm{x}.\mathcal{S}[\![e]\!] \mathbin{@} k) \mathbin{@} k & \\
\textsc{throw}\ e_1\ e_2 : k & = & e_1 : \lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \mathbin{@} \mathrm{v} & \text{if } e_1 \text{ is not a value} \\
\textsc{throw}\ v_1\ e_2 : k & = & e_2 : \mathcal{S}_t[\![v_1]\!] & \text{if } e_2 \text{ is not a value} \\
\textsc{throw}\ v_1\ v_2 : k & = & \mathcal{S}_t[\![v_1]\!] \mathbin{@} \mathcal{S}_t[\![v_2]\!] &
\end{array}
$$

$$
\begin{array}{rcll}
\mathrm{x} : \mathrm{E} & = & \mathrm{E}[\mathrm{x}] & \\
\mathrm{c} : \mathrm{E} & = & \mathrm{E}[\mathrm{c}] & \\
\textsc{fun}^{\mathrm{N}}\ \mathrm{f}\ \mathrm{x}.e : \mathrm{E} & = & \mathrm{E}[\![\textsc{fun}\ \mathrm{f}\ \mathrm{x}.\mathcal{S}[\![e]\!]]\!] & \\
\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e : \mathrm{E} & = & \mathrm{E}[\![\textsc{fun}\ \mathrm{f}\ \mathrm{x}.\mathcal{S}_t[\![e]\!]]\!] & \\
(e_1 \mathbin{@}^{\mathrm{T}} e_2)^{\mathrm{T}} : \mathrm{E} & = & e_1 : \mathrm{E}\big[([\,] \mathbin{@}^{\mathrm{T}} \mathcal{S}_t[\![e_2]\!])^{\mathrm{T}}\big] & \text{if } e_1 \text{ is not a value} \\
(v_1 \mathbin{@}^{\mathrm{T}} e_2)^{\mathrm{T}} : \mathrm{E} & = & e_2 : \mathrm{E}\big[(\mathcal{S}_t[\![v_1]\!] \mathbin{@}^{\mathrm{T}} [\,])^{\mathrm{T}}\big] & \text{if } e_2 \text{ is not a value} \\
(v_1 \mathbin{@}^{\mathrm{T}} v_2)^{\mathrm{T}} : \mathrm{E} & = & \mathrm{E}\big[(\mathcal{S}_t[\![v_1]\!] \mathbin{@}^{\mathrm{T}} \mathcal{S}_t[\![v_2]\!])^{\mathrm{T}}\big] &
\end{array}
$$

Fig. 7. The selective colon translation on expressions

annotated expressions, so we extend it to work on continuation values, $\langle \mathrm{E} \rangle$, which are values and as such trivial.

$$\langle \mathrm{E}' \rangle : \mathrm{E} \;=\; \mathrm{E}[\mathrm{E}' : \mathrm{id}]$$

where $\mathrm{id} = \lambda \mathrm{x}.\mathrm{x}$ and $\mathrm{E} : k$ defines either a continuation function or a context as displayed in Figure 8, in which E^{T} represents any non-empty context with a top-most annotation as trivial.

The $\mathrm{E} : k$-translation yields either continuation functions or contexts, depending on the annotation of the innermost levels of the context argument,

$$
\begin{aligned}
[\,] : k &= k \\
\mathrm{E}^{\mathrm{T}} : k &= \mathrm{E}^{\mathrm{T}} : [k \mathbin{@} [\,]] \\
(\mathrm{E} \mathbin{@}^{\mathrm{N}} e_2)^{\mathrm{N}} : k &= \mathrm{E} : \lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \mathbin{@} (\lambda \mathrm{v}'.\mathrm{v} \mathbin{@} \mathrm{v}' \mathbin{@} k) \\
(\mathrm{E} \mathbin{@}^{\mathrm{T}} e_2)^{\mathrm{N}} : k &= \mathrm{E} : \lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \mathbin{@} (\lambda \mathrm{v}'.k \mathbin{@} (\mathrm{v} \mathbin{@} \mathrm{v}')) \\
(v_1 \mathbin{@}^{\mathrm{N}} \mathrm{E})^{\mathrm{N}} : k &= \mathrm{E} : (\lambda \mathrm{v}'.\mathcal{S}_t[\![v_1]\!] \mathbin{@} \mathrm{v}' \mathbin{@} k) \\
(v_1 \mathbin{@}^{\mathrm{T}} \mathrm{E})^{\mathrm{N}} : k &= \mathrm{E} : (\lambda \mathrm{v}'.k \mathbin{@} (\mathcal{S}_t[\![v_1]\!] \mathbin{@} \mathrm{v}')) \\
\textsc{throw}\ \mathrm{E}\ e_2 : k &= \mathrm{E} : \lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \mathbin{@} \mathrm{v} \\
\textsc{throw}\ v_1\ \mathrm{E} : k &= \mathrm{E} : \mathcal{S}_t[\![v_1]\!]
\end{aligned}
$$

$$
\begin{aligned}
[\,] : \mathrm{E} &= \mathrm{E} \\
(\mathrm{E} \mathbin{@}^{\mathrm{T}} e_2)^{\mathrm{T}} : \mathrm{E}' &= \mathrm{E} : \mathrm{E}'[[\,] \mathbin{@} \mathcal{S}_t[\![e_2]\!]] \\
(v_1 \mathbin{@}^{\mathrm{T}} \mathrm{E})^{\mathrm{T}} : \mathrm{E}' &= \mathrm{E} : \mathrm{E}'[\mathcal{S}_t[\![v_1]\!] \mathbin{@} [\,]]
\end{aligned}
$$

Fig. 8. The selective colon translation on contexts.

and the E : E-translation always gives a context, but requires that the first argument's outermost annotation is trivial.

These colon-translations satisfy a number of correspondences.

Proposition 4.2 *For all contexts* E_1, E_2, *and* E_3, *and continuation functions (closed functional values) the following equalities hold.*

$$
\begin{aligned}
\mathrm{E}_1[\mathrm{E}_2[\,]] : k &= \mathrm{E}_2 : (\mathrm{E}_1 : k) \\
\mathrm{E}_1[\mathrm{E}_2[\,]] : \mathrm{E}_3 &= \mathrm{E}_2 : (\mathrm{E}_1 : \mathrm{E}_3)
\end{aligned}
$$

Proof. The proof is by simple induction on the context E_1.

- If $\mathrm{E}_1 = [\,]$ then $(\mathrm{E}_1[\mathrm{E}_2[\,]] : k) = (\mathrm{E}_2 : k) = (\mathrm{E}_2 : (\mathrm{E}_1 : k))$ and $(\mathrm{E}_1[\mathrm{E}_2[\,]] : \mathrm{E}_3) = (\mathrm{E}_2 : \mathrm{E}_3) = (\mathrm{E}_2 : (\mathrm{E}_1 : \mathrm{E}_3))$.
- If $\mathrm{E}_1 = \left[(\mathrm{E} \mathbin{@}^{\mathrm{N}} e_2)^{\mathrm{N}}\right]$ then

$$
\begin{aligned}
\mathrm{E}_1[\mathrm{E}_2[\,]] : k &= (\mathrm{E}[\mathrm{E}_2] \mathbin{@}^{\mathrm{N}} e_2)^{\mathrm{N}} : k \\
&= \mathrm{E}[\mathrm{E}_2] : \lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \mathbin{@} (\lambda \mathrm{v}'.\mathrm{v} \mathbin{@} \mathrm{v}' \mathbin{@} k) && \text{(def. of E : } k\text{)} \\
&= \mathrm{E}_2 : (\mathrm{E} : \lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \mathbin{@} (\lambda \mathrm{v}'.\mathrm{v} \mathbin{@} \mathrm{v}' \mathbin{@} k)) && \text{(I.H.)} \\
&= \mathrm{E}_2 : (\left[(\mathrm{E} \mathbin{@}^{\mathrm{N}} e_2)^{\mathrm{N}}\right] : k) && \text{(def. E : } k\text{)}
\end{aligned}
$$

- The remaining cases are similar.

□

One would expect that similar equalities hold for the colon translations on expressions, i.e., $\mathrm{E}[e] : k = e : (\mathrm{E} : k)$ and $\mathrm{E}[e] : \mathrm{E}' = e : (\mathrm{E} : \mathrm{E}')$, and indeed these equalities hold in most cases. The exception is when E is non-empty and the "innermost" expression of the context is not annotated as trivial, e.g., $\mathrm{E}_1[([\,] \mathbin{@} e_1)^{\mathrm{N}}]$ for some context E_1 and expression e_1, and e is a value. Normally the $e : k$ translation descends the left-hand side and rebuilds the context on the right hand side, either as a continuation function or as a context, depending on the annotation. The exception mentioned, $\mathrm{E}[e] : k$, the focus of the colon translation, the expression on the left hand side of the colon, would never descend all the way down to a value. We have made special cases for $v \mathbin{@} e$ to bypass administrative reductions, so $\mathrm{E}[v] : k$ would not equal $v : (\mathrm{E} : k)$, because the latter introduces an administrative reduction. Reducing that administrative reduction, applying k to $\mathcal{S}_t[\![v]\!]$, does lead to $v : (\mathrm{E} : k)$ again in one or more reduction steps. That is, if e is a value and E is not a trivial context then $e : (\mathrm{E} : k) = (\mathrm{E} : k) \mathbin{@} \mathcal{S}_t[\![e]\!] \to^* \mathrm{E}[e] : k$, and likewise for the $e : \mathrm{E}$-relation.

Proposition 4.3 *For all contexts* E *and* E′*, expressions* e*, and continuation functions* k

$$e : (\mathrm{E} : k) \to^* \mathrm{E}[e] : k$$

$$e : (\mathrm{E} : \mathrm{E}') \to^* \mathrm{E}[e] : \mathrm{E}' \quad (\textit{if } e \textit{ trivial})$$

and $\to^*$ *is* $\to^0$*, i.e., equality, if* e *is not a value.*

Proof. Omitted. □

4.2 Colon-translation lemmas

Plotkin used four lemmas to prove his simulation and indifference theorems. We only prove simulation, which corresponds to Plotkin's simulation, since we already know that indifference does not hold for a selective CPS transformation (at least unless the selectivity is based on the effect of nontermination as well).

Lemma 4.4 (Substitution) *If* $\Gamma[\mathrm{x} : \tau_1] \vdash e : \tau_2, \mathrm{A}$ *and* $\vdash v : \tau_1, \mathrm{T}$ *is a closed value then*

$$\mathcal{S}[\![e]\!]\,[\mathcal{S}_t[\![v]\!]/\mathrm{x}] = \mathcal{S}[\![e\,[v/\mathrm{x}]]\!]$$

$$\mathcal{S}_t[\![e]\!]\,[\mathcal{S}_t[\![v]\!]/\mathrm{x}] = \mathcal{S}_t[\![e\,[v/\mathrm{x}]]\!] \ (\textit{if } e \textit{ is trivial})$$

Proof. The proof is by induction on the structure of e, using the distributive properties of substitution and taking the trivial cases before the non-trivial ones (because the $\mathcal{S}[\![\cdot]\!]$ translation defers trivial subexpressions to the $\mathcal{S}_t$ transformation). The details have been omitted. □

Lemma 4.5 (Initial reduction) *If* $\Gamma \vdash e : \tau, \mathrm{A}$ *and* k *is a continuation function of appropriate type then*

$$\begin{aligned} \mathcal{S}[\![e]\!] \mathbin{@} k &\to^* e : k \\ \mathrm{E}[\mathcal{S}_t[\![e]\!]] &= e : \mathrm{E} \ (\textit{if } e \textit{ is trivial}) \end{aligned}$$

Proof. Again, the proof is by induction on the structure of e with the $\mathcal{S}[\![\cdot]\!]$ case taken after the $\mathcal{S}_t$ case for trivial expressions.

The $\mathrm{E}[\mathcal{S}_t[\![\cdot]\!]] = \cdot : \mathrm{E}$ **case:** There are four cases covering all trivial expressions:

- If e is a value or an identifier then $e : \mathrm{E} = \mathrm{E}[\mathcal{S}_t[\![e]\!]]$ by definition of $e : \mathrm{E}$.
- If $e = (e_1 \mathbin{@}^{\mathrm{T}} e_2)^{\mathrm{T}}$ (e_1 not a value) then

$$\begin{aligned} (e_1 \mathbin{@}^{\mathrm{T}} e_2)^{\mathrm{T}} : \mathrm{E} &= e_1 : \mathrm{E}[[\,] \mathbin{@} \mathcal{S}_t[\![e_2]\!]] && \text{(def. } e : \mathrm{E}) \\ &= \mathrm{E}[\mathcal{S}_t[\![e_1]\!] \mathbin{@} \mathcal{S}_t[\![e_2]\!]] && \text{(I.H.)} \\ &= \mathrm{E}\big[\mathcal{S}_t[\![(e_1 \mathbin{@}^{\mathrm{T}} e_2)^{\mathrm{T}}]\!]\big] && \text{(def. } \Psi) \end{aligned}$$

- If $e = (v_1 \mathbin{@}^{\mathrm{T}} e_2)^{\mathrm{T}}$ (e_2 not a value) then

$$\begin{aligned} (v_1 \mathbin{@}^{\mathrm{T}} e_2)^{\mathrm{T}} : \mathrm{E} &= e_2 : \mathrm{E}[\mathcal{S}_t[\![v_1]\!] \mathbin{@} [\,]] && \text{(def. } e : \mathrm{E}) \\ &= \mathrm{E}[\mathcal{S}_t[\![v_1]\!] \mathbin{@} \mathcal{S}_t[\![e_2]\!]] && \text{(I.H.)} \\ &= \mathrm{E}\big[\mathcal{S}_t[\![(v_1 \mathbin{@}^{\mathrm{T}} e_2)^{\mathrm{T}}]\!]\big] && \text{(def. } \Psi) \end{aligned}$$

- If $e = (v_1 \mathbin{@}^{\mathrm{T}} v_2)^{\mathrm{T}}$ then

$$\begin{aligned} (v_1 \mathbin{@}^{\mathrm{T}} v_2)^{\mathrm{T}} : \mathrm{E} &= \mathrm{E}[\mathcal{S}_t[\![v_1]\!] \mathbin{@} \mathcal{S}_t[\![v_2]\!]] && \text{(I.H.)} \\ &= \mathrm{E}\big[\mathcal{S}_t[\![(v_1 \mathbin{@}^{\mathrm{T}} v_2)^{\mathrm{T}}]\!]\big] && \text{(def. } \Psi) \end{aligned}$$

This accounts for all trivial expressions.

The $\mathcal{S}[\![\cdot]\!] \mathbin{@} k \to^* \cdot : k$ **case:** There is one sub-case for each non-trivial expression, and one case for all trivial expressions:

- If e is trivial then $\mathcal{S}[\![e]\!] \mathbin{@} k = k \mathbin{@} \mathcal{S}_t[\![e]\!] = e : [k \mathbin{@} [\,]] = e : k$ from the above cases and the definition of $e : k$.
- If $e = (e_1 \mathbin{@}^{\mathrm{N}} e_2)^{\mathrm{N}}$ (e_1 not a value) then

$$\begin{aligned} &\mathcal{S}[\![(e_1 \mathbin{@}^{\mathrm{N}} e_2)^{\mathrm{N}}]\!] \mathbin{@} k \\ &\to \mathcal{S}[\![e_1]\!] \mathbin{@} (\lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \mathbin{@} (\lambda \mathrm{v}'.\mathrm{v} \mathbin{@} \mathrm{v}' \mathbin{@} k)) && \text{(def. } \mathcal{S}[\![\cdot]\!]) \\ &\to^* e_1 : (\lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \mathbin{@} (\lambda \mathrm{v}'.\mathrm{v} \mathbin{@} \mathrm{v}' \mathbin{@} k)) && \text{(I.H.)} \\ &= (e_1 \mathbin{@}^{\mathrm{N}} e_2)^{\mathrm{N}} : k && \text{(def. } e : k) \end{aligned}$$

- If $e = (v_1 \ @^{\mathrm{N}} \ e_2)^{\mathrm{N}}$ (e_2 not a value) then

$$\begin{aligned}
&\mathcal{S}[\![(v_1 \ @^{\mathrm{N}} \ e_2)^{\mathrm{N}}]\!] \ @ \ k \\
&\quad \to \ \mathcal{S}[\![v_1]\!] \ @ \ (\lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \ @ \ (\lambda \mathrm{v}'.\mathrm{v} \ @ \ \mathrm{v}' \ @ \ k)) && \text{(def. } \mathcal{S}[\![\cdot]\!]) \\
&\quad \to \ (\lambda \mathrm{v}.\mathcal{S}[\![e_2]\!] \ @ \ (\lambda \mathrm{v}'.\mathrm{v} \ @ \ \mathrm{v}' \ @ \ k)) \ @ \ \mathcal{S}_t[\![v_1]\!] && \text{(def. } \mathcal{S}[\![v]\!]) \\
&\quad \to \ \mathcal{S}[\![e_2]\!] \ @ \ (\lambda \mathrm{v}'.\mathcal{S}_t[\![v_1]\!] \ @ \ \mathrm{v}' \ @ \ k) \\
&\quad \to^* \ e_2 : \lambda \mathrm{v}'.\mathcal{S}_t[\![v_1]\!] \ @ \ \mathrm{v}' \ @ \ k && \text{(I.H.)} \\
&\quad = \ (v_1 \ @^{\mathrm{N}} \ e_2)^{\mathrm{N}} : k && \text{(def. } e : k)
\end{aligned}$$

- If $e = (v_1 \ @^{\mathrm{N}} \ v_2)^{\mathrm{N}}$ then the proof is similar to the previous case except two values need to be applied to continuations instead of just one.
- If $e = (e_1 \ @^{\mathrm{T}} \ e_2)^{\mathrm{N}}$ then the proofs are similar to the ones for $e = (e_1 \ @^{\mathrm{T}} \ e_2)^{\mathrm{N}}$ except that the innermost application is $k \ @ \ (\mathrm{v} \ @ \ \mathrm{v}')$ instead of $(\mathrm{v} \ @ \ \mathrm{v}') \ @ \ k$.
- If $e = \textsc{callcc} \ \mathrm{x}.e_1$ then $\mathcal{S}[\![\textsc{callcc} \ \mathrm{x}.e_1]\!] \ @ \ k \to (\lambda \mathrm{x}.\mathcal{S}[\![e_1]\!] \ @ \ k) \ @ \ k = \textsc{callcc} \ \mathrm{x}.e_1 : k$.
- If $e = \textsc{throw} \ e_1 \ e_2$ the proofs are similar to the ones for application.

□

Lemma 4.6 (Simulation) *If* $\mathrm{E}[\![e]\!] \to \mathrm{E}'[\![e']\!]$ *is one of the reduction rules for the annotated language, then*

$$\mathrm{E}[\![e]\!] : \mathrm{id} \to^* \mathrm{E}'[\![e']\!] : \mathrm{id}$$

and if the reduction is not of a THROW *expression, then the* $\to^*$ *is actually one or more steps.*

Proof. Counting annotations, there are five cases:

- If $e = (\textsc{fun}^{\mathrm{N}} \ \mathrm{f} \ \mathrm{x}.e_1 \ @^{\mathrm{N}} \ v)^{\mathrm{N}}$ then

$$\begin{aligned}
&\mathrm{E}\big[(\textsc{fun}^{\mathrm{N}} \ \mathrm{f} \ \mathrm{x}.e_1 \ @^{\mathrm{N}} \ v)^{\mathrm{N}}\big] : \mathrm{id} \\
&\quad = \ (\textsc{fun}^{\mathrm{N}} \ \mathrm{f} \ \mathrm{x}.e_1 \ @^{\mathrm{N}} \ v)^{\mathrm{N}} : (\mathrm{E} : \mathrm{id}) && \text{(Prop. 4.3)} \\
&\quad = \ \mathcal{S}_t[\![\textsc{fun}^{\mathrm{N}} \ \mathrm{f} \ \mathrm{x}.e_1]\!] \ @ \ \mathcal{S}_t[\![v]\!] \ @ \ (\mathrm{E} : \mathrm{id}) && \text{(def. } e : k) \\
&\quad = \ \textsc{fun} \ \mathrm{f} \ \mathrm{x}.\mathcal{S}[\![e_1]\!] \ @ \ \mathcal{S}_t[\![v]\!] \ @ \ (\mathrm{E} : \mathrm{id}) && \text{(def. } \Psi) \\
&\quad \to \ \mathcal{S}[\![e_1]\!] \ \big[\mathcal{S}_t[\![\textsc{fun}^{\mathrm{N}} \ \mathrm{f} \ \mathrm{x}.e_1]\!]/\mathrm{f}\big] \ [\mathcal{S}_t[\![v]\!]/\mathrm{x}] \ @ \ (\mathrm{E} : \mathrm{id}) \\
&\quad = \ \mathcal{S}[\![e_1 \ \big[\textsc{fun}^{\mathrm{N}} \ \mathrm{f} \ \mathrm{x}.e_1/\mathrm{f}\big] \ [v/\mathrm{x}]]\!] \ @ \ (\mathrm{E} : \mathrm{id}) && \text{(Lemma 4.4)} \\
&\quad \to^* \ e_1 \ \big[\textsc{fun}^{\mathrm{N}} \ \mathrm{f} \ \mathrm{x}.e_1/\mathrm{f}\big] \ [v/\mathrm{x}] : (\mathrm{E} : \mathrm{id}) && \text{(Lemma 4.5)} \\
&\quad \to^* \ \mathrm{E}\big[e_1 \ \big[\textsc{fun}^{\mathrm{N}} \ \mathrm{f} \ \mathrm{x}.e_1/\mathrm{f}\big] \ [v/\mathrm{x}]\big] : \mathrm{id} && \text{(Prop. 4.3)}
\end{aligned}$$

- If $e = (\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1\ @^{\mathrm{T}}\ v)^{\mathrm{N}}$ then we know that e_1 is trivial, since otherwise the function would be annotated N, and $\mathrm{E} : k$ is a continuation since E has no trivial inner sub-contexts.

$$
\begin{array}{lll}
\mathrm{E}\big[(\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1\ @^{\mathrm{T}}\ v)^{\mathrm{N}}\big] : \mathrm{id} & & \\
\quad = \ (\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1\ @^{\mathrm{T}}\ v)^{\mathrm{N}} : (\mathrm{E} : \mathrm{id}) & & \text{(Prop. 4.3)} \\
\quad = \ (\mathrm{E} : \mathrm{id}) \ @\ (\mathcal{S}_t[\![\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1]\!] \ @\ \mathcal{S}_t[\![v]\!]) & & \text{(def. } e : k\text{)} \\
\quad = \ (\mathrm{E} : \mathrm{id}) \ @\ (\textsc{fun}\ \mathrm{f}\ \mathrm{x}.\mathcal{S}_t[\![e_1]\!] \ @\ \mathcal{S}_t[\![v]\!]) & & \text{(def. } \Psi\text{)} \\
\quad \to \ (\mathrm{E} : \mathrm{id}) \ @\ \mathcal{S}_t[\![e_1]\!]\ \big[\mathcal{S}_t[\![\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1]\!]/\mathrm{f}\big]\ [\mathcal{S}_t[\![v]\!]/\mathrm{x}] & & \\
\quad = \ (\mathrm{E} : \mathrm{id}) \ @\ \mathcal{S}_t[\![e_1\ \big[\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1/\mathrm{f}\big]\ [v/\mathrm{x}]]\!] & & \text{(Lemma 4.4)} \\
\quad \to^* \ e_1\ \big[\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1/\mathrm{f}\big]\ [v/\mathrm{x}] : [(\mathrm{E} : \mathrm{id}) \ @\ [\,]] & & \text{(Lemma 4.5, } e_1 \text{ trivial)} \\
\quad = \ e_1\ \big[\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1/\mathrm{f}\big]\ [v/\mathrm{x}] : (\mathrm{E} : \mathrm{id}) & & \text{(def. } e : k\text{)} \\
\quad \to^* \ \mathrm{E}\big[e_1\ \big[\textsc{fun}^{\mathrm{N}}\ \mathrm{f}\ \mathrm{x}.e_1/\mathrm{f}\big]\ [v/\mathrm{x}]\big] : \mathrm{id} & & \text{(Prop. 4.3)}
\end{array}
$$

- If $e = (e_1\ @^{\mathrm{T}}\ e_2)^{\mathrm{T}}$ then either $\mathrm{E} : k$ is a context or a continuation. If it is a continuation the proof proceeds just as the previous case. If it is a context then

$$
\begin{array}{lll}
\mathrm{E}\big[(\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1\ @^{\mathrm{T}}\ v)^{\mathrm{T}}\big] : \mathrm{id} & & \\
\quad = \ (\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1\ @^{\mathrm{T}}\ v)^{\mathrm{T}} : (\mathrm{E} : \mathrm{id}) & & \text{(Prop. 4.3)} \\
\quad = \ (\mathrm{E} : \mathrm{id})\big[\mathcal{S}_t[\![\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1]\!] \ @\ \mathcal{S}_t[\![v]\!]\big] & & \text{(def. } e : k\text{)} \\
\quad = \ (\mathrm{E} : \mathrm{id})[\![\textsc{fun}\ \mathrm{f}\ \mathrm{x}.\mathcal{S}_t[\![e_1]\!] \ @\ \mathcal{S}_t[\![v]\!]] & & \text{(def. } \Psi\text{)} \\
\quad \to \ (\mathrm{E} : \mathrm{id})\big[\mathcal{S}_t[\![e_1]\!]\ \big[\mathcal{S}_t[\![\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1]\!]/\mathrm{f}\big]\ [\mathcal{S}_t[\![v]\!]/\mathrm{x}]\big] & & \\
\quad = \ (\mathrm{E} : \mathrm{id})\big[\mathcal{S}_t[\![e_1\ \big[\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1/\mathrm{f}\big]\ [v/\mathrm{x}]]\!]\big] & & \text{(Lemma 4.4)} \\
\quad \to^* \ e_1\ \big[\textsc{fun}^{\mathrm{T}}\ \mathrm{f}\ \mathrm{x}.e_1/\mathrm{f}\big]\ [v/\mathrm{x}] : (\mathrm{E} : \mathrm{id}) & & \text{(Lemma 4.5, } e_1 \text{ trivial)} \\
\quad \to^* \ \mathrm{E}\big[e_1\ \big[\textsc{fun}^{\mathrm{N}}\ \mathrm{f}\ \mathrm{x}.e_1/\mathrm{f}\big]\ [v/\mathrm{x}]\big] : \mathrm{id} & & \text{(Prop. 4.3)}
\end{array}
$$

- If $e = \text{CALLCC x.}e_1$ then

$$\begin{aligned}
&\mathrm{E}[\![\text{CALLCC x.}e_1]\!] : \mathrm{id} \\
&= \text{CALLCC x.}e_1 : (\mathrm{E} : \mathrm{id}) && \text{(Prop. 4.3)} \\
&= (\lambda \mathrm{x}.\mathcal{S}[\![e_1]\!] \mathbin{@} (\mathrm{E} : \mathrm{id})) \mathbin{@} (\mathrm{E} : \mathrm{id}) && \text{(def. } e : k\text{)} \\
&\to \mathcal{S}[\![e_1]\!] \mathbin{@} (\mathrm{E} : \mathrm{id})\,[(\mathrm{E} : \mathrm{id})/\mathrm{x}] \\
&= \mathcal{S}[\![e_1]\!]\,[(\mathrm{E} : \mathrm{id})/\mathrm{x}] \mathbin{@} (\mathrm{E} : \mathrm{id}) && \text{(E : } k \text{ is closed)} \\
&= \mathcal{S}[\![e_1]\!]\,[\mathcal{S}_t[\![\langle \mathrm{E} \rangle]\!]/\mathrm{x}] \mathbin{@} (\mathrm{E} : \mathrm{id}) && \text{(def. } \mathcal{S}_t[\![\langle \mathrm{E} \rangle]\!]\text{)} \\
&= \mathcal{S}[\![e_1\,[\langle \mathrm{E} \rangle/\mathrm{x}]]\!] \mathbin{@} (\mathrm{E} : \mathrm{id}) && \text{(Lemma 4.4)} \\
&\to^* e_1\,[\langle \mathrm{E} \rangle/\mathrm{x}] : (\mathrm{E} : \mathrm{id}) && \text{(Lemma 4.5)} \\
&\to^* \mathrm{E}[\![e_1\,[\langle \mathrm{E} \rangle/\mathrm{x}]]\!] : \mathrm{id} && \text{(Prop. 4.3)}
\end{aligned}$$

- If $e = \text{THROW } \langle \mathrm{E}' \rangle\ v$ then

$$\begin{aligned}
\mathrm{E}[\![\text{THROW } \langle \mathrm{E}' \rangle\ v]\!] : \mathrm{id} &= \mathcal{S}_t[\![\langle \mathrm{E}' \rangle]\!] \mathbin{@} \mathcal{S}_t[\![v]\!] && \text{(def. } e : k\text{)} \\
&= (\mathrm{E}' : \mathrm{id}) \mathbin{@} \mathcal{S}_t[\![v]\!] && \text{(def. } \mathcal{S}_t[\![\langle \mathrm{E} \rangle]\!]\text{)} \\
&= v : [(\ @\ \mathrm{E}' : \mathrm{id})[\ ]] && \text{(def. } v : \mathrm{E}\text{)} \\
&= v : (\mathrm{E}' : \mathrm{id}) && \text{(def. } (e)^{\mathrm{T}} : k\text{)} \\
&\to^* \mathrm{E}'[\![v]\!] : \mathrm{id} && \text{(Prop. 4.3)}
\end{aligned}$$

In all cases except THROW, there is at least one reduction step.

□

4.3 Proof of correctness

To prove the correctness of the selective CPS transformation, we use the simulation lemma in two ways.

Proof. The proof of $e \to^* \mathrm{c} \implies \mathcal{S}[\![e]\!] \mathbin{@} \mathrm{id} \to^* \mathrm{c}$ follows directly from the lemma 4.5 and repeated use of lemma 4.6. Assume $e \to^* \mathrm{c}$.

$$\begin{aligned}
\mathcal{S}[\![e]\!] \mathbin{@} \mathrm{id} &\to^* e : \mathrm{id} && \text{(Lemma 4.5)} \\
&\to^* \mathrm{c} : \mathrm{id} && \text{(Lemma 4.6, repeated)} \\
&= \mathrm{c} : [\mathrm{id} \mathbin{@} [\ ]] && \text{(def. } (e)^{\mathrm{T}} : k\text{)} \\
&= \mathrm{id} \mathbin{@} \mathrm{c} && \text{(def. c : E)} \\
&\to \mathrm{c}
\end{aligned}$$

.

The other direction of correctness, $\mathcal{S}[\![e]\!]$ @ id $\rightarrow^*$ c $\implies$ $e \rightarrow^*$ c, is shown by contraposition. Assuming that for no c does $e \rightarrow^*$ c, that is, e diverges, allows us to show that the same holds for $\mathcal{S}[\![e]\!]$.

The proof that transformation preserves divergence also follow from Lemmas 4.5 and 4.6. Since $\mathcal{S}[\![e]\!]$ @ id $\rightarrow^*$ e : id it suffices to show that e : id has an arbitrary long reduction sequence.

Assume that e diverges. We show that for any n there exists an m such that if $e \rightarrow^m e_1$ then $(e : \mathrm{id}) \rightarrow^* (e_1 : \mathrm{id})$ in n or more reduction steps.

This is proven by induction on n. The base case ($n = 0$) is trivial. For the induction case ($n + 1$) look at the n case. There exists m such that $e \rightarrow^m e_1$ and $e : \mathrm{id} \rightarrow^* e_1 : \mathrm{id}$. Look at the reduction sequence from e_1.

- If the first reduction step ($e_1 \rightarrow e_2$) is not the reduction of a THROW expression, then $e_1 : \mathrm{id} \rightarrow^+ e_2 : \mathrm{id}$, and $m + 1$ gives us our $n + 1$ or longer reduction sequence of e : id.
- If the first reduction step ($e_1 \rightarrow e_2$) is of a THROW expression, then $e_1 : \mathrm{id} \rightarrow^* e_2 : \mathrm{id}$. In that case we look at the next step in the same way. Either we find a reduction that is not a THROW, and we get the m needed for the proof, or there is nothing but reductions of THROW expressions in the infinite reduction sequence of e_1.

  There can not be an infinite sequence of reductions of THROW expressions, since reducing a THROW expression necessarily reduces the size of the entire program. A substitution into a context corresponds to the application of a linear function, and it reduces the size of the expression if one counts it as, e.g., number of distinct subexpressions or number of THROW-expressions.

That means that e : id has an infinite reduction sequence. □

5 Conclusion

We have proven the correctness of a selective CPS transformation based on an effect analysis. Similar proofs can be made for other λ-encodings and computational effects (e.g., with monads), where the immediate choice would be the effect of non-termination. This is the effect that is encoded by the traditional CPS transformation of languages with no other effects, and if one has an annotation of such a program, marking terminating (effect-free) expressions to keep in direct style, then the method works just as well.

5.1 Perspectives

Danvy and Hatcliff's CPS transformation after strictness analysis [6] generalizes the call-by-name and the call-by-value CPS transformations. The same authors' CPS transformation after totality analysis [7] generalizes the call-by-name CPS transformation and the identity transformation. In the same manner, the present work generalizes the call-by-value CPS transformation

and the identity transformation, and proves this generalization correct.

Danvy and Filinski introduced the one-pass CPS-transformation [5] that removes the administrative reductions from the result by performing them at transformation time. This optimization can be applied to the selective CPS-transformation presented here as well. A proof of the correctness of the one-pass CPS-transformation also using Plotkin's colon translation exists [8]. We expect that the methods used for proving correctness of the selective- and the one-pass CPS transformations are orthogonal, and can easily be combined.

The selective CPS transformation presented here is based on an effect analysis and should generalize to other computational effects than control, e.g., state or I/O. The proof will not carry over to other effects, since it relies on the choice of λ-encoding of the effect primitives, but we expect that the structure of the proof can be preserved.

The approach taken is "Curry-style" in the sense that we have given a language and its operational meaning, and only after the fact we have associated types and effect annotation to the untyped terms. A "Church-style" approach, such as Filinski's [10,11], would have defined the language with explicit types and effect annotation, so that only well-typed, consistently annotated programs are given a semantics.

5.2 Future work

It is possible to prove results similar to the present ones for other choices of effects and combinations of effects. A sensible choice would be a monadic effect of state and control, since it is sufficient to implement all other choices of layered monads [11]. A proof similar to the present one for both state and control effects would be a logical next step.

Acknowledgments:
The method of extending the colon translation to selective CPS transformation was originally developed in cooperation with Jung-taek Kim and Kwangkeun Yi from KAIST in Korea, and with Olivier Danvy from BRICS in Denmark. The present work would not have been possible without their inspiration. Thanks are also due to Andrzej Filinski and to the anonymous referees for their comments.

References

[1] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, 1992.

[2] Hans-J. Boehm, editor. *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994. ACM Press.

[3] William Clinger, Daniel P. Friedman, and Mitchell Wand. A scheme for a higher-level semantic algebra. In John Reynolds and Maurice Nivat, editors, *Algebraic Methods in Semantics*, pages 237–250. Cambridge University Press, 1985.

[4] Daniel Damian and Olivier Danvy. Syntactic accidents in program analysis. In Philip Wadler, editor, *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming*, pages 209–220, Montréal, Canada, September 2000. ACM Press.

[5] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[6] Olivier Danvy and John Hatcliff. CPS transformation after strictness analysis. *ACM Letters on Programming Languages and Systems*, 1(3):195–212, 1993.

[7] Olivier Danvy and John Hatcliff. On the transformation between direct and continuation semantics. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Proceedings of the 9th Conference on Mathematical Foundations of Programming Semantics*, number 802 in Lecture Notes in Computer Science, pages 627–648, New Orleans, Louisiana, April 1993. Springer-Verlag.

[8] Olivier Danvy and Lasse R. Nielsen. A higher-order colon translation. In Herbert Kuchen and Kazunori Ueda, editors, *Fifth International Symposium on Functional and Logic Programming*, number 2024 in Lecture Notes in Computer Science, pages 78–91, Tokyo, Japan, March 2001. Springer-Verlag. Extended version available as the technical report BRICS RS-00-33.

[9] Matthias Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Department of Computer Science, Indiana University, Bloomington, Indiana, August 1987.

[10] Andrzej Filinski. Representing monads. In Boehm [2], pages 446–457.

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

[12] Robert Harper, Bruce F. Duba, and David MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, October 1993.

[13] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Boehm [2], pages 458–471.

[14] Jung-taek Kim and Kwangkeun Yi. Interconnecting Between CPS Terms and Non-CPS Terms. In Sabry [20].

[15] Jung-taek Kim, Kwangkeun Yi, and Olivier Danvy. Assessing the overhead of ML exceptions by selective CPS transformation. In Greg Morrisett, editor, *Record of the 1998 ACM SIGPLAN Workshop on ML and its Applications*, Baltimore, Maryland, September 1998. Also appears as BRICS technical report RS-98-15.

[16] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, California, June 1989. IEEE Computer Society Press.

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

[18] John Reppy. Local CPS conversion in a direct-style compiler. In Sabry [20].

[19] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972).

[20] Amr Sabry, editor. *Proceedings of the Third ACM SIGPLAN Workshop on Continuations CW'01*, number 545 in Technical Report, Computer Science Department, Indiana University, Bloomington, Indiana, December 2000.

[21] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.