Improving CPS-Based Partial Evaluation: Writing Cogen by Hand

Anders Bondorf*
DIKU
Department of Computer Science
Universitetsparken 1
DK-2100 Copenhagen Ø, Denmark
anders@diku.dk

Dirk Dussart**
Departement Computerwetenschappen
Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Leuven (Heverlee), Belgium
Dirk.Dussart@cs.kuleuven.ac.be

Abstract

It is well-known that self-applicable partial evaluation can be used to generate compiler generators: $\mathit{cogen} = \mathit{mix}(\mathit{mix}, \mathit{mix})$, where *mix* is the specializer (partial evaluator). However, writing *cogen* by hand gives several advantages: (1) Contrasting to when writing a self-applicable *mix*, one is not restricted to write *cogen* in the same language as it treats [HL91]. (2) A handwritten *cogen* can be more efficient than a *cogen* generated by self-application; in particular, a handwritten *cogen* typically performs no (time consuming) environment manipulations whereas one generated by self-application does. (3) When working in statically typed languages with user defined data types, the self-application approach requires *encoding* data type values [Bon88, Lau91, DNBV91], resulting in relatively inefficient (*cogen*-generated) compilers that spend much of their time on coding and decoding. By writing *cogen* by hand, the coding problem is eliminated [HL91, BW93].

Specializers written in *continuation passing style* (abbreviated "cps") perform better than specializers written in direct style (abbreviated "ds") [Bon92]. For example, a specializer written in cps straightforwardly handles non-unfoldable let-expressions with static body.

The contribution of this paper is to combine the idea of hand-writing *cogen* with cps-based specialization. We develop a handwritten cps-*cogen* which is superior to a ds-*cogen* for the same reason that a cps-specializer is superior to a ds-specializer: the cps-*cogen* can for example handle non-unfoldable let-expressions with static body. Hand-writing a cps-*cogen* is done along the same lines as hand-writing a ds-*cogen*, but some additional non-standard two-level η-expansions turn out to be needed.

The handwritten cps-*cogen* presented here is efficient in that it performs continuation processing (β-reductions of continuation applications) already at compiler-generation time. Only some continuation processing can be done at compiler generation time, however, so the resulting programs generated by *cogen* also contain continuations.

We prove our handwritten cps-*cogen* correct with respect to a cps-specializer. We also give a correctness proof of a handwritten ds-*cogen*; this proof is much simpler than the cps-proof, but to the best of our knowledge, no handwritten ds-*cogen* has been proved correct before.

1 Introduction

Cps-based specializers are more powerful than ds-based specializers. For example, a cps-specializer straightforwardly specializes `((let y=... in λx.x+x+y) 7)` into `(let y=... in 14+y)` when the let-expression is non-unfoldable. The cps-specializer is able to do so because it explicitly manipulates a context: a cps-specializer is able to move the context "apply to 7" across the let-binding into the let-body.

In this paper we show how to hand-write a cps-based *cogen*. We derive the handwritten cps-*cogen* from a (handwritten) cps-specializer. However, to make it easier to follow the derivation, we first show how to derive (and prove correctness of) a handwritten ds-*cogen* $\mathcal{C}_d$ from a (handwritten) ds-specializer $\mathcal{S}_d$: the ds-based *cogen* is much simpler to derive than the cps-based *cogen*. Then we derive and prove correctness of the handwritten cps-*cogen* $\mathcal{C}_{cp}$ from a (handwritten) cps-specializer $\mathcal{S}_{cp}$. See the horizontal arrows in Figure 1.

$$\begin{array}{ccc} \text{ds-specializer } \mathcal{S}_d & \longrightarrow & \text{ds-}\mathit{cogen}\ \mathcal{C}_d \\ \downarrow & & \downarrow \\ \text{cps-specializer } \mathcal{S}_{cp} & \longrightarrow & \text{cps-}\mathit{cogen}\ \mathcal{C}_{cp} \end{array}$$

Figure 1: Overview

The cps-specializer $\mathcal{S}_{cp}$ can be derived from the ds-specializer $\mathcal{S}_d$ (the leftmost vertical arrow in Figure 1) [Bon92]. We shall derive the cps-*cogen* $\mathcal{C}_{cp}$ from the cps-specializer $\mathcal{S}_{cp}$. In Section 4 we briefly discuss how to derive $\mathcal{C}_{cp}$ from $\mathcal{C}_d$ instead (rightmost vertical arrow); this derivation is relevant if one is to hand-write a cps-*cogen* for a language where a handwritten ds-*cogen* already exists.

We shall consider specialization similar to the one of ***Lambda-mix*** [GJ91]. In this paper we only consider a source language consisting of the *strict* (call-by-value) weak-head normal form pure lambda calculus (variables, λ-abstraction and application) extended with a let-construct, see Figure 2.

*Current postal address: Computer Resources International A/S, Bregnerødvej 144, DK-3460 Birkerød, Denmark; e-mail: use anders@diku.dk

**Funded by the National Fund for Scientific Research (Belgium). This work was done during two stays at DIKU in Copenhagen, 1993; DIKU and K.U. Leuven supported Dirk Dussart's visits to DIKU.

We include the let-construct in the source language to cover a form that cps-based specialization treats better than ds-specialization does [Bon92].

$$
\begin{array}{l}
\mathit{Variable} = \mathit{String}\,; \quad e \in \mathit{Expression}'\,;\ v \in \mathit{Variable} \\
e \ ::= \ \mathit{Var}\,v \ \mid \ \mathit{Lam}\,v\,e_1 \ \mid \ \mathit{App}\,e_1\ e_2 \ \mid \ \mathit{Let}\,v\,e_1\ e_2
\end{array}
$$

Figure 2: Abstract syntax of source language

In an extended version of the paper, we will also cover the remaining constructs from Lambda-mix (constants, conditionals and *fix*), as well as primitive operations and operations on tuples. Conditionals are interesting as a cps-specializer, contrasting to a ds-specializer, is able to handle conditionals with dynamic test but static branches [Bon92]. Operations on tuples are interesting as they illustrate the coding problem that arises when writing a specializer *mix*, but not when hand-writing *cogen*. Tuples are as easy to handle in a handwritten cps-*cogen* as in a handwritten ds-*cogen*: no particular problems with tuples arise due to cps.

When hand-writing *cogen*, we shall need some abstract syntax constructors in addition to *Var*, *Lam*, *App* and *Let*. These additional constructors are *Var*◇, *Fresh*, *Lam*◇, *App*◇ and *Let*◇. The semantics of the source language, extended with these additional forms, is given in Figure 3. The meta-language used in this paper is strict: λ- and *let*-forms are thus strict as well as environment updates $\rho[\ldots \mapsto \ldots]$. Notice that *fresh*() generates a fresh variable name (a string) and that the forms *Lam*◇, *App*◇ and *Let*◇ are used to generate expressions rather than values as *Lam*, *App* and *Let* do.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathcal{E} : \mathit{Expression} \times (\mathit{Variable} \rightarrow \mathit{Value}) \rightarrow \mathit{Value}} \\
\mathcal{E}[\![\mathit{Var}\,v]\!]\rho & = & \rho\ v \\
\mathcal{E}[\![\mathit{Lam}\,v\,e_1]\!]\rho & = & \lambda w\,.\,\mathcal{E}[\![e_1]\!]\rho[v \mapsto w] \\
\mathcal{E}[\![\mathit{App}\,e_1\ e_2]\!]\rho & = & (\mathcal{E}[\![e_1]\!]\rho)\,(\mathcal{E}[\![e_2]\!]\rho) \\
\mathcal{E}[\![\mathit{Let}\,v\,e_1\ e_2]\!]\rho & = & \mathcal{E}[\![e_2]\!]\rho[v \mapsto \mathcal{E}[\![e_1]\!]\rho] \\
\mathcal{E}[\![\mathit{Var}\diamond\,v]\!]\rho & = & \mathit{Var}\,(\rho\ v) \\
\mathcal{E}[\![\mathit{Fresh}]\!]\rho & = & \mathit{fresh}() \\
\mathcal{E}[\![\mathit{Lam}\diamond\,e_1\ e_2]\!]\rho & = & \mathit{Lam}\,(\mathcal{E}[\![e_1]\!]\rho)\,(\mathcal{E}[\![e_2]\!]\rho) \\
\mathcal{E}[\![\mathit{App}\diamond\,e_1\ e_2]\!]\rho & = & \mathit{App}\,(\mathcal{E}[\![e_1]\!]\rho)\,(\mathcal{E}[\![e_2]\!]\rho) \\
\mathcal{E}[\![\mathit{Let}\diamond\,e_1\ e_2\ e_3]\!]\rho & = & \mathit{Let}\,(\mathcal{E}[\![e_1]\!]\rho)\,(\mathcal{E}[\![e_2]\!]\rho)\,(\mathcal{E}[\![e_3]\!]\rho)
\end{array}
$$

Figure 3: Semantics of extended source language

Programs to be partially evaluated will be annotated and written in a ***two-level*** language [NN88, GJ91]. The two-level language is specified in Figure 4. Each of the compound forms now exist in two versions, a static version (e.g. $\mathit{Lam}\,v\,t_1$) and a dynamic version (e.g. $\underline{\mathit{Lam}}\,v\,t_1$). The static versions will be reduced at partial evaluation time, and code will be emitted for the dynamic versions.

It turns out to be helpful for cps-based specialization that all source expression variables have distinct names. In the rest of this paper, variable t therefore only ranges over two-level expressions where all variables names are different (variables names can always be made distinct by α-conversion).

Only programs that are ***well-annotated*** may be specialized. Type rules for checking well-annotatedness are given

$$
\begin{array}{l}
t \in \mathit{2Expression};\ v \in \mathit{Variable} \\
t \ ::= \ \mathit{Var}\,v \ \mid \ \mathit{Lam}\,v\,t_1 \ \mid \ \mathit{App}\,t_1\ t_2 \ \mid \ \mathit{Let}\,v\,t_1\ t_2 \ \mid \\
\qquad \underline{\mathit{Lam}}\,v\,t_1 \ \mid \ \underline{\mathit{App}}\,t_1\ t_2 \ \mid \ \underline{\mathit{Let}}\,v\,t_1\ t_2
\end{array}
$$

Figure 4: Syntax of two-level language

in [GJ91] (not for the let-form, though, but it is simple to add). Annotating programs can be done automatically by ***binding-time analysis***, see e.g. [Gom90, Hen91].

2 Direct style

Figure 5 specifies the ds-specializer $\mathcal{S}_d$. Specializer $\mathcal{S}_d$ is a part of the Lambda-mix specializer T from Appendix A of the paper [GJ91], extended with (straightforward) rules for the static and dynamic let-forms. Notice that domain *2Value* is equal to domain *Value* since *Value* already includes the forms generated when evaluating the forms *Lam*◇, *App*◇ and *Let*◇ (Figure 3).

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathcal{S}_d : \mathit{2Expression} \times (\mathit{Variable} \rightarrow \mathit{2Value}) \rightarrow \mathit{2Value}} \\
\mathcal{S}_d[\![\mathit{Var}\,v]\!]\rho & = & \rho\ v \\
\mathcal{S}_d[\![\mathit{Lam}\,v\,t_1]\!]\rho & = & \lambda w\,.\,\mathcal{S}_d[\![t_1]\!]\rho[v \mapsto w] \\
\mathcal{S}_d[\![\mathit{App}\,t_1\ t_2]\!]\rho & = & (\mathcal{S}_d[\![t_1]\!]\rho)\,(\mathcal{S}_d[\![t_2]\!]\rho) \\
\mathcal{S}_d[\![\mathit{Let}\,v\,t_1\ t_2]\!]\rho & = & \mathcal{S}_d[\![t_2]\!]\rho[v \mapsto \mathcal{S}_d[\![t_1]\!]\rho] \\
\mathcal{S}_d[\![\underline{\mathit{Lam}}\,v\,t_1]\!]\rho & = & \mathit{let}\ n{=}\mathit{fresh}() \\
 & & \mathit{in}\ \mathit{Lam}\,n\,(\mathcal{S}_d[\![t_1]\!]\rho[v \mapsto \mathit{Var}\,n]) \\
\mathcal{S}_d[\![\underline{\mathit{App}}\,t_1\ t_2]\!]\rho & = & \mathit{App}\,(\mathcal{S}_d[\![t_1]\!]\rho)\,(\mathcal{S}_d[\![t_2]\!]\rho) \\
\mathcal{S}_d[\![\underline{\mathit{Let}}\,v\,t_1\ t_2]\!]\rho & = & \mathit{let}\ n{=}\mathit{fresh}() \\
 & & \mathit{in}\ \mathit{Let}\,n\,(\mathcal{S}_d[\![t_1]\!]\rho)\,(\mathcal{S}_d[\![t_2]\!]\rho[v \mapsto \mathit{Var}\,n])
\end{array}
$$

Figure 5: Ds-specializer

Notice that ds-specializer $\mathcal{S}_d$ cannot specialize forms such as $t = \mathit{App}\,(\underline{\mathit{Let}}\,v_1\ \ldots\ (\mathit{Lam}\,v_2\ \ldots))\,(\mathit{Var}\,v_3)$ as $\mathcal{S}_d$ requires the body of a $\underline{\mathit{Let}}$-form to specialize to an expression: the result of $\mathcal{S}_d$'s call $\mathcal{S}_d[\![t_2]\!]\rho[v \mapsto \mathit{Var}\,n]$ must be an expression as it is an argument to the abstract syntax constructor *Let*. But $\mathcal{S}_d$ specializes $\mathit{Lam}\,v_2\ \ldots$ to a function $\lambda w.\ldots$, not to an expression, so expression t is not well-annotated with respect to $\mathcal{S}_d$. To specialize the expression, the annotations should be $\underline{\mathit{App}}\,(\underline{\mathit{Let}}\,v_1\ \ldots\ (\underline{\mathit{Lam}}\,v_2\ \ldots))\,(\mathit{Var}\,v_3)$ (as it also follows from the well-annotatedness rules of [GJ91]); being underlined, the application would consequently not be β-reduced by $\mathcal{S}_d$ during specialization.

We now present a ds-*cogen* $\mathcal{C}_d$ derived from the ds-specializer $\mathcal{S}_d$; see Figure 6. Essentially, instead of ***performing*** what $\mathcal{S}_d$ does, compiler generator $\mathcal{C}_d$ ***generates code*** that will perform the same operations ***when evaluated*** (by $\mathcal{E}$). For example, specializer $\mathcal{S}_d$ performs an application when treating *App*-forms, but $\mathcal{C}_d$ generates an *App*-expression which, when evaluated, performs an application. And, where $\mathcal{S}_d$ generates an *App*-expression when treating $\underline{\mathit{App}}$-forms, compiler generator $\mathcal{C}_d$ generates an *App*◇-expression which, when evaluated, generates an *App*-expression.

Notice that $\mathcal{C}_d$ takes no environment (ρ) argument. Avoiding environment manipulation is possible by reusing source variable names in the treatments of *Lam*, *Let*, $\underline{\mathit{Lam}}$

$$
\begin{array}{lcl}
\mathcal{C}_d : \mathit{2Expression} \rightarrow \mathit{Expression} & & \\
\mathcal{C}_d[\![\mathit{Var}\, v]\!] & = & \mathit{Var}\, v \\
\mathcal{C}_d[\![\mathit{Lam}\, v\, t_1]\!] & = & \mathit{Lam}\, v\, (\mathcal{C}_d[\![t_1]\!]) \\
\mathcal{C}_d[\![\mathit{App}\, t_1\ t_2]\!] & = & \mathit{App}\, (\mathcal{C}_d[\![t_1]\!])\, (\mathcal{C}_d[\![t_2]\!]) \\
\mathcal{C}_d[\![\mathit{Let}\, v\, t_1\ t_2]\!] & = & \mathit{Let}\, v\, (\mathcal{C}_d[\![t_1]\!])\, (\mathcal{C}_d[\![t_2]\!]) \\
\mathcal{C}_d[\![\underline{\mathit{Lam}}\, v\, t_1]\!] & = & \mathit{Let}\, m\, \mathit{Fresh}\, (\mathit{Let}\, v\, (\mathit{Var}\diamond\, m)\, (\mathit{Lam}\diamond\, (\mathit{Var}\, m)\, (\mathcal{C}_d[\![t_1]\!]))) \\
\mathcal{C}_d[\![\underline{\mathit{App}}\, t_1\ t_2]\!] & = & \mathit{App}\diamond\, (\mathcal{C}_d[\![t_1]\!])\, (\mathcal{C}_d[\![t_2]\!]) \\
\mathcal{C}_d[\![\underline{\mathit{Let}}\, v\, t_1\ t_2]\!] & = & \mathit{Let}\, m\, \mathit{Fresh}\, (\mathit{Let}\, v\, (\mathit{Var}\diamond\, m)\, (\mathit{Let}\diamond\, (\mathit{Var}\, m)\, (\mathcal{C}_d[\![t_1]\!])\, (\mathcal{C}_d[\![t_2]\!])))
\end{array}
$$

Figure 6: Ds-cogen

and $\underline{\mathit{Let}}$ (notice e.g. how $\mathcal{S}_d$'s *Lam*-rule $\lambda w\,.\,\mathcal{S}_d[\![t_1]\!]\rho[v \mapsto w]$ turns into $\mathit{Lam}\, v\, (\mathcal{C}_d[\![t_1]\!])$ in $\mathcal{C}_d$: source name v is used instead of w whereby the binding $[v \mapsto w]$ can be ignored), but it is non-trivial to see that this does not lead to unexpected name clashes. The reason is briefly that $\mathcal{C}_d$ performs no symbolic unfolding and thus preserves the scoping structure of the source program. The handwritten compiler generators [HL91, BW93] did not manipulate environments either (but no correctness proofs were given there). Compiler generators generated by self-application do manipulate environments (see e.g. [GJ91]) and thus they are less efficient than the handwritten ones.

The following theorem states that the handwritten *cogen* $\mathcal{C}_d$ is indeed correct with respect to the specializer $\mathcal{S}_d$ (and in particular this also proves that the environment-free treatment of variables in $\mathcal{C}_d$ is correct). The theorem states that evaluating the code generated by $\mathcal{C}_d$ in environment ρ yields the same result as specializing by $\mathcal{S}_d$ (in environment ρ):

Theorem 1 (Correctness of ds-*cogen*)

$\forall t, \rho : \mathcal{E}[\![\mathcal{C}_d[\![t]\!]]\!]\rho = \mathcal{S}_d[\![t]\!]\rho$

Proof: By structural induction over two-level expressions. See Appendix A.1 for details. □

3 Continuation passing style

Figure 7 contains a cps-specializer $\mathcal{S}_{cp}$, derived from $\mathcal{S}_d$ by (non-standard) cps-transformation as described in [Bon92]; continuation ι is the identity continuation $\lambda x\,.\,x$. The cps-specializer $\mathcal{S}_{cp}$ is more powerful than the ds-specializer $\mathcal{S}_d$: it does not constrain the annotations of the body of $\underline{\mathit{Let}}$-forms (the type rule for checking well-annotatedness for $\underline{\mathit{Let}}$-forms is consequently more liberal for cps-based specialization than for ds-specialization). For example, specializer $\mathcal{S}_{cp}$ is able to specialize the form $\mathit{App}\, (\underline{\mathit{Let}}\, v_1\ \ldots\ (\mathit{Lam}\, v_2\ \ldots))\, (\mathit{Var}\, v_3)$, hence β-reducing the application during specialization (contrasting to $\mathcal{S}_d$, cf. Section 2).

Notice that the identity continuation ι is used not only to initialize, but also when treating $\underline{\mathit{Lam}}$-forms. This non-standard "impure" form of cps turns out to be necessary to allow the desired liberal treatment of $\underline{\mathit{Let}}$-forms, propagating κ "over the let-binding". The more pure cps-code $\mathit{let}\, n{=}\mathit{fresh}()\ \mathit{in}\ \mathcal{S}_{cp}[\![t_1]\!]\rho[v \mapsto n](\lambda x\,.\,\kappa\,(\mathit{Lam}\, n\, x))$ that one might have expected in the $\underline{\mathit{Lam}}$-rule thus gives an *incorrect* result if the lambda-body t_1 is a $\underline{\mathit{Let}}$-form. Indeed, the let- and λ-bindings are reversed. In short, the problem is that continuations that dump their argument in the body-position of a generated lambda-expression are not allowed to be propagated over the binding when specializing $\underline{\mathit{Let}}$-forms; the continuation $\lambda x\,.\,\kappa\,(\mathit{Lam}\, n\, x)$ is such a disallowed form. The code in Figure 7 does not contain any such "ill-behaved" continuations. We refer to [Bon92] for further details.

We are now ready to present the handwritten cps-*cogen* $\mathcal{C}_{cp}$, see Figure 8. Compiler generator $\mathcal{C}_{cp}$ is derived in the same way from $\mathcal{S}_{cp}$ as $\mathcal{C}_d$ was derived from $\mathcal{S}_d$: instead of *performing* what $\mathcal{S}_{cp}$ does, $\mathcal{C}_{cp}$ generates code that will perform the same operations when evaluated. Deriving the $\mathcal{C}_{cp}$-rules for *Lam* and *App* involves some additional steps that have no analogue in the $\mathcal{C}_d$-derivation; these steps will be described below. Notice that similarly to $\mathcal{C}_d$, compiler generator $\mathcal{C}_{cp}$ performs no operations on environments, contrasting to what a compiler generator generated by self-application would do. Also notice that $\mathcal{C}_{cp}$ has a continuation argument: we want $\mathcal{C}_{cp}$ to perform continuation reductions already at *cogen*-time rather than suspending all continuation processing to appear in the programs generated by *cogen* (such a simpler cps-*cogen* can be written, but it is certainly less interesting).

We shall now explain why the *Lam*- and *App*-rules look the way they do. At a first try, we might optimistically have written the *Lam*- and *App*-rules in the following more "natural" way:

$$
\begin{array}{lcl}
\mathcal{C}_{cp}[\![\mathit{Lam}\, v\, t_1]\!]\kappa & = & \kappa\, (\mathit{Lam}\, v\, (\mathcal{C}_{cp}[\![t_1]\!])) \\
\mathcal{C}_{cp}[\![\mathit{App}\, t_1\ t_2]\!]\kappa & = & \mathcal{C}_{cp}[\![t_1]\!](\lambda x\,.\,\mathcal{C}_{cp}[\![t_2]\!](\lambda y\,.\,\mathit{App}\, (\mathit{App}\, x\, y)\, \kappa))
\end{array}
$$

Let us first consider the incorrect *Lam*-rule. Notice that $\mathcal{C}_{cp}[\![t_1]\!]$ is a *function* (from continuations to expressions) whereas the second argument to constructor *Lam* must be an *expression* of type *Expression*. We can fix this problem by a special two-level η-expansion that converts a function to an expression (a λ-form into a *Lam*-form): $f \mapsto \mathit{Lam}\, n\, (f(\mathit{Var}\, n))$ where n is fresh to avoid name shadowing. Instead of $\mathcal{C}_{cp}[\![t_1]\!]$, we would thus write $\mathit{Lam}\, n\, (\mathcal{C}_{cp}[\![t_1]\!](\mathit{Var}\, n))$. But now there is a problem with the expression $\mathcal{C}_{cp}[\![t_1]\!](\mathit{Var}\, n)$ as $\mathcal{C}_{cp}$'s second argument must be a function (a continuation), not an expression such as $\mathit{Var}\, n$. We therefore perform another kind of two-level η-expansion, this time converting an expression into a function: $e \mapsto \lambda x\,.\,\mathit{App}\, e\, x$. We then obtain $\mathcal{C}_{cp}[\![t_1]\!](\lambda x\,.\,\mathit{App}\, (\mathit{Var}\, n)\, x)$. The *Lam*-rule of Figure 8 has now emerged.

In a similar way, the *App*-rule of Figure 8 is obtained from the incorrect one by η-expanding κ in the incorrect expression $\mathit{App}\, (\mathit{App}\, x\, y)\, \kappa$ into $\mathit{Lam}\, n\, (\kappa\, (\mathit{Var}\, n))$; *App*'s second argument must be an expression, not a function.

$$
\begin{array}{lcl}
\mathcal{S}_{cp} : \mathit{2Expression} \times (\mathit{Variable} \to \mathit{2Value}) \times (\mathit{2Value} \to \mathit{2Value}) \to \mathit{2Value} & & \\
\mathcal{S}_{cp}[\![\mathit{Var}\,v]\!]\rho\kappa & = & \kappa\,(\rho\,v) \\
\mathcal{S}_{cp}[\![\mathit{Lam}\,v\,t_1]\!]\rho\kappa & = & \kappa\,(\lambda w\,.\,\mathcal{S}_{cp}[\![t_1]\!]\rho[v \mapsto w]) \\
\mathcal{S}_{cp}[\![\mathit{App}\,t_1\;t_2]\!]\rho\kappa & = & \mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda x\,.\,\mathcal{S}_{cp}[\![t_2]\!]\rho(\lambda y\,.\,(x\,y)\,\kappa)) \\
\mathcal{S}_{cp}[\![\mathit{Let}\,v\,t_1\;t_2]\!]\rho\kappa & = & \mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda x\,.\,\mathcal{S}_{cp}[\![t_2]\!]\rho[v \mapsto x]\kappa) \\
\mathcal{S}_{cp}[\![\underline{\mathit{Lam}}\,v\,t_1]\!]\rho\kappa & = & \kappa\,(\mathit{let}\,n\mathord{=}\mathit{fresh}()\;\mathit{in}\;\mathit{Lam}\,n\,(\mathcal{S}_{cp}[\![t_1]\!]\rho[v \mapsto \mathit{Var}\,n]\iota)) \\
\mathcal{S}_{cp}[\![\underline{\mathit{App}}\,t_1\;t_2]\!]\rho\kappa & = & \mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda x\,.\,\mathcal{S}_{cp}[\![t_2]\!]\rho(\lambda y\,.\,\kappa\,(\mathit{App}\,x\,y))) \\
\mathcal{S}_{cp}[\![\underline{\mathit{Let}}\,v\,t_1\;t_2]\!]\rho\kappa & = & \mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda x\,.\,\mathit{let}\,n\mathord{=}\mathit{fresh}()\;\mathit{in}\;\mathit{Let}\,n\,x\,(\mathcal{S}_{cp}[\![t_2]\!]\rho[v \mapsto \mathit{Var}\,n]\kappa))
\end{array}
$$

Figure 7: Cps-specializer

$$
\begin{array}{lcl}
\mathcal{C}_{cp} : \mathit{2Expression} \times (\mathit{Expression} \to \mathit{Expression}) \to \mathit{Expression} & & \\
\mathcal{C}_{cp}[\![\mathit{Var}\,v]\!]\kappa & = & \kappa\,(\mathit{Var}\,v) \\
\mathcal{C}_{cp}[\![\mathit{Lam}\,v\,t_1]\!]\kappa & = & \kappa\,(\mathit{Lam}\,v\,(\mathit{let}\,n\mathord{=}\mathit{fresh}()\;\mathit{in}\;\mathit{Lam}\,n\,(\mathcal{C}_{cp}[\![t_1]\!](\lambda x\,.\,\mathit{App}\,(\mathit{Var}\,n)\,x)))) \\
\mathcal{C}_{cp}[\![\mathit{App}\,t_1\;t_2]\!]\kappa & = & \mathcal{C}_{cp}[\![t_1]\!](\lambda x\,.\,\mathcal{C}_{cp}[\![t_2]\!](\lambda y\,.\,\mathit{App}\,(\mathit{App}\,x\,y)\,(\mathit{let}\,n\mathord{=}\mathit{fresh}()\;\mathit{in}\;\mathit{Lam}\,n\,(\kappa\,(\mathit{Var}\,n))))) \\
\mathcal{C}_{cp}[\![\mathit{Let}\,v\,t_1\;t_2]\!]\kappa & = & \mathcal{C}_{cp}[\![t_1]\!](\lambda x\,.\,\mathit{Let}\,v\,x\,(\mathcal{C}_{cp}[\![t_2]\!]\kappa)) \\
\mathcal{C}_{cp}[\![\underline{\mathit{Lam}}\,v\,t_1]\!]\kappa & = & \kappa\,(\mathit{Let}\,m\,\mathit{Fresh}\,(\mathit{Let}\,v\,(\mathit{Var}\diamond\,m)\,(\mathit{Lam}\diamond\,(\mathit{Var}\,m)\,(\mathcal{C}_{cp}[\![t_1]\!]\iota)))) \\
\mathcal{C}_{cp}[\![\underline{\mathit{App}}\,t_1\;t_2]\!]\kappa & = & \mathcal{C}_{cp}[\![t_1]\!](\lambda x\,.\,\mathcal{C}_{cp}[\![t_2]\!](\lambda y\,.\,\kappa\,(\mathit{App}\diamond\,x\,y))) \\
\mathcal{C}_{cp}[\![\underline{\mathit{Let}}\,v\,t_1\;t_2]\!]\kappa & = & \mathcal{C}_{cp}[\![t_1]\!](\lambda x\,.\,\mathit{Let}\,m\,\mathit{Fresh}\,(\mathit{Let}\,v\,(\mathit{Var}\diamond\,m)\,(\mathit{Let}\diamond\,(\mathit{Var}\,m)\,x\,(\mathcal{C}_{cp}[\![t_2]\!]\kappa))))
\end{array}
$$

Figure 8: Cps-cogen

The η-expansions used here resemble the η-conversions used in [DF92] to separate "administrative" from "non-administrative" continuations in cps-transformation. Also, similar η-conversions were used for ***binding-time improvements*** in [Bon91].

We note that expression $\mathit{Lam}\,n\,(\kappa\,(\mathit{Var}\,n))$ in the *App*-rule generates continuations that are present in the programs generated by $\mathcal{C}_{cp}$. Thus, even though $\mathcal{C}_{cp}$ performs continuation processing (β-reductions), it also generates code that still contains (some) continuation processing. This is again analogue to the distinction between "administrative" and "non-administrative" continuations in cps-transformations: only administrative continuations can be β-reduced during cps-transformation.

To prove correctness of $\mathcal{C}_{cp}$ with respect to $\mathcal{S}_{cp}$, we must prove the following: for all t and ρ, it holds that $\mathcal{E}[\![\mathcal{C}_{cp}[\![t]\!]\iota]\!]\rho = \mathcal{S}_{cp}[\![t]\!]\rho\iota$. That is, evaluating the expression generated by $\mathcal{C}_{cp}$ in some environment ρ gives the same result as specializing t in the same environment. Both $\mathcal{C}_{cp}$ and $\mathcal{S}_{cp}$ are initially called with the identity continuation ι. To prove this equality inductively, we need a more general theorem that holds not only when the continuations are ι. Can we hope to simply replace ι by κ and then expect that the equality holds for all κ? The answer is unfortunately "no". The reason is simple: the type of $\mathcal{S}_{cp}$'s continuation parameter is $\mathit{2Value} \to \mathit{2Value}$ whereas the type of $\mathcal{C}_{cp}$'s continuation parameter is $\mathit{Expression} \to \mathit{Expression}$. However, given a $\mathcal{C}_{cp}$-type continuation κ, we can construct a $\mathcal{S}_{cp}$-type continuation: $\lambda a\,.\,\mathit{let}\,m\mathord{=}\mathit{fresh}()\;\mathit{in}\;\mathcal{E}[\![\kappa\,(\mathit{Var}\,m)]\!]\rho[m \mapsto a]$. The idea here is to evaluate the expression generated by applying κ to an argument, taking care not to evaluate a which already is a $\mathit{2Value}$ (this is the reason why the continuation is ***not*** simply $\lambda a\,.\,\mathcal{E}[\![\kappa\,a]\!]\rho$). This leads to the following correctness theorem.

THEOREM 2 (Correctness of cps-*cogen*)

$\forall t, \rho, \kappa : \mathcal{E}[\![\mathcal{C}_{cp}[\![t]\!]\kappa]\!]\rho =$
$\quad \mathcal{S}_{cp}[\![t]\!]\rho(\lambda a\,.\,\mathit{let}\;m\mathord{=}\mathit{fresh}()\;\mathit{in}\;\mathcal{E}[\![\kappa\,(\mathit{Var}\,m)]\!]\rho[m \mapsto a])$

PROOF: By structural induction over two-level expressions. See Appendix A.2 for details. $\square$

In this theorem, as well as in Appendix A.2, we implicitly assume some restrictions on κ when quantifying by $\forall t, \ldots, \kappa \ldots$: continuation κ must be related to two-level expression t in the sense that κ only ranges over those continuations that are generated when computing $\mathcal{C}_{cp}[\![t_1]\!]\iota$ where t is a subexpression of t_1. That is, we only consider the ***relevant*** continuations, not all continuations. Notice that the identity continuation ι is a relevant continuation (possible value for κ).
The desired correctness property now follows as a corollary:

COROLLARY 3 (Correctness of cps-*cogen*)

$\forall t, \rho : \mathcal{E}[\![\mathcal{C}_{cp}[\![t]\!]\iota]\!]\rho = \mathcal{S}_{cp}[\![t]\!]\rho\iota$

PROOF: Follows from Theorem 2 since

$$
\begin{array}{l}
\lambda a\,.\,\mathit{let}\,m\mathord{=}\mathit{fresh}()\;\mathit{in}\;\mathcal{E}[\![\iota\,(\mathit{Var}\,m)]\!]\rho[m \mapsto a] \overset{\beta}{=} \\
\quad \lambda a\,.\,\mathit{let}\,m\mathord{=}\mathit{fresh}()\;\mathit{in}\;\mathcal{E}[\![\mathit{Var}\,m]\!]\rho[m \mapsto a] \overset{\mathcal{E}}{=} \\
\quad \lambda a\,.\,\mathit{let}\,m\mathord{=}\mathit{fresh}()\;\mathit{in}\;a \overset{\text{Lemma 8}}{=} \lambda a\,.\,a = \iota
\end{array}
$$
$\square$

(Lemma 8 can be found in Appendix A.2.) In the proof of Theorem 2, a number of lemmas are used; these are found in Appendix A.2. It is worth noticing that the lemmas only hold when t and κ are restricted as described earlier: all variable names in t must be distinct (α-conversion, cf. Section 1), and κ must be relevant.

4 Deriving $\mathcal{C}_{cp}$ from $\mathcal{C}_d$

In retrospect, when comparing $\mathcal{C}_d$ and $\mathcal{C}_{cp}$, we notice that $\mathcal{C}_{cp}$ could have been derived from $\mathcal{C}_d$ rather than from $\mathcal{S}_{cp}$: by cps-transforming the $\mathcal{C}_d$, taking into account to use the non-standard cps $\underline{Lam}$-rule, and performing appropriate η-expansions for the *Lam*- and *App*-rules. This way of deriving $\mathcal{S}_{cp}$ might be useful in a context where a handwritten ds-*cogen* already exists, for example if one were to write a cps-*cogen* for the ML-*cogen* described in [BW93]. We believe that this can be done without great difficulty.

5 Related work

Already in the REDFUN-project was a *cogen* for a subset of Lisp written by hand [BHOS76]. The motivation was that the specializer could not be self-applied.

In [Hol89], a handwritten *cogen* was based on macro expansion. In the paper [HL91], a ds-*cogen* for a statically typed language is described. The ideas from [HL91] were used for hand-writing a ds-*cogen* for a subset of Standard ML [BW93].

Quite recently the work by Lawall and Danvy in [LD94] came to our attention. Lawall and Danvy show how the cps-specializer from [Bon92] can be almost automatically derived from a ds-specializer by inserting the control operators shift and reset (see [DF90]) at selected places and cps converting the resulting specialiser. They also devote some attention to how their ideas could be used in the context of a handwritten cogen.

6 Conclusion

We have demonstrated how an efficient cps-based *cogen* can be written by hand. The handwritten *cogen* performs no environment manipulations, contrasting to *cogen*s generated by self-applying specializers. The cps-*cogen* is derived naturally from a cps-specializer, except that some non-standard η-expansions are needed in the treatment of *Lam*- and *App*-forms to shift between functions and expressions. We have given correctness proofs for the cps-*cogen* as well as for a ds-*cogen*.

We believe that our handwritten *cogen* is a good starting point for hand-writing cps-based *cogen*s for larger strict functional languages. Our work does not immediately carry over to lazy languages as the cps-transformation we have used is the strict cps-transformation. However, it is plausible that a similar development could be made for a lazy language using call-by-name cps-transformation (with loss of sharing as a consequence).

Acknowledgements

We would like to thank Neil Jones, Torben Mogensen, Julia Lawall for the fruitful discussions on the subject; also thanks to Karel De Vlaminck and Eddy Bevers for his indispensable contributions in the final stages of the paper.

A Proofs of the theorems 1 and 2

Both proofs are by induction over t; the case analysis is over the syntactic forms specified in Figure 4. All equalities are annotated to explain why equality holds. Notice that β- and η-equalities are used: β/η do not in general hold for the typed ($\mathcal{C}_d$ and $\mathcal{S}_d$ are both simply typed) strict weak-head normal form lambda-calculus. β/η thus only hold when termination properties do not change; we only use β/η when this is the case. We use β-abstraction to prevent duplicating expressions of form ***fresh***(). Also notice that in both proofs we rely on the fact that the variable m, introduced in the $\underline{Lam}$- and $\underline{Let}$-rule in both ds- and cps-*cogen*, is unique: m does not occur in input programs and can not be generated by application of ***fresh***(). By construction it is assured that $\forall t_1$: neither $\mathcal{C}_{cp}[\![t_1]\!]\kappa$ (where κ is relevant) nor $\mathcal{C}_d[\![t_1]\!]$ contains m as a free variable, nor that any definition of m shadows another definition of m (see Figure 6 and Figure 8).

A.1 Proof of Theorem 1

See Figure 9.

A.2 Proof of Theorem 2

We first give the lemmas needed for the inductive proof of Theorem 2. Notice that Lemma 8 was also used in the proof of Corollary 3. We use M and E to range over meta-expressions (as opposed to e that ranges over object expressions). Recall (Section 3) that only two-level expressions t with all variable names distinct and only well-behaved continuations κ are considered when quantifying over t and κ.

LEMMA 4 (Environment simplification)

$\forall t, \kappa$: *if* v is bound in t *then*

$$\forall \rho : \textit{let}\ m{=}\textit{fresh}()\ \textit{in}\ \mathcal{E}[\![\kappa\,(\,\textit{Var}\,m)]\!]\rho[v \mapsto \ldots] \;=\; \textit{let}\ m{=}\textit{fresh}()\ \textit{in}\ \mathcal{E}[\![\kappa\,(\,\textit{Var}\,m)]\!]\rho$$

that is, term $\kappa(\textit{Var}\,m)$ will not contain any free occurrences of v.

PROOF: Continuation κ is generated independently of t, so when applied to $(\textit{Var}\,m)$ it cannot (since all source variable names are distinct) generate expressions with any (and hence no free) v-occurrences. □

LEMMA 5 (Extracting out κ's argument)

$\forall t, \kappa$: *if* t is one of the forms $\textit{Var}\,v$, $\textit{Lam}\,v\,t_1$, $\underline{\textit{Lam}}\,v\,t_1$ or $\underline{\textit{App}}\,t_1\,t_2$ *then*, when computing $\mathcal{C}_{cp}[\![t]\!]\kappa$, the following equality holds for (all relevant instances of) the expressions $\kappa\,E$ in the right-hand sides of the sides of the rules for *Var*, *Lam*, $\underline{\textit{Lam}}$ and $\underline{\textit{App}}$:

$$\forall \rho : \mathcal{E}[\![\kappa\,E]\!]\rho \;=\; (\lambda a\,.\ \textit{let}\ m{=}\textit{fresh}()\ \textit{in}\ \mathcal{E}[\![\kappa\,(\,\textit{Var}\,m)]\!]\rho[m \mapsto a])\,(\mathcal{E}[\![E]\!]\rho)$$

PROOF: First notice that since $\rho[m \mapsto a]$ is strict in a, we may β-reduce $(\lambda a.\ \ldots)(\mathcal{E}[\![E]\!]\rho)$. We thus have to prove $\mathcal{E}[\![\kappa\,E]\!]\rho \;=\; \textit{let}\ m{=}\textit{fresh}()\ \textit{in}\ \mathcal{E}[\![\kappa\,(\,\textit{Var}\,m)]\!]\rho[m \mapsto \mathcal{E}[\![E]\!]\rho]$. We shall refer to the left- and rigth-hand sides of this equality as lhs and rhs below.

Let e be the value of (meta-)expression E, let e_1 be the value of (meta-)expression $\kappa\,E$, and let e_2 be the value of (meta-)expression $\kappa\,(\textit{Var}\,m)$; notice from the type of κ (Figure 8) that the values e, e_1 and e_2 are all expressions. It then holds that e_1 always contains at least one leaf which is a copy of e, and this leaf is always placed in a strict position, i.e. when evaluating e_1, e is guaranteed also to be evaluated ("evaluation" is done by $\mathcal{E}$); apart from the e-leaves, the rest of e_1 is independent of e. These properties of e_1 are easily inductively proved by considering all possible relevant continuations κ.

$$\mathcal{E}\llbracket\mathcal{C}_d\llbracket \mathit{Var}\,v\rrbracket\rrbracket\rho \stackrel{\mathcal{C}_d}{=} \mathcal{E}\llbracket \mathit{Var}\,v\rrbracket\rho \stackrel{\mathcal{E}}{=} \rho\,v \stackrel{\mathcal{S}_d}{=} \mathcal{S}_d\llbracket \mathit{Var}\,v\rrbracket\rho.$$

$$\mathcal{E}\llbracket\mathcal{C}_d\llbracket \mathit{Lam}\,v\,t_1\rrbracket\rrbracket\rho \stackrel{\mathcal{C}_d}{=} \mathcal{E}\llbracket \mathit{Lam}\,v\,(\mathcal{C}_d\llbracket t_1\rrbracket)\rrbracket\rho \stackrel{\mathcal{E}}{=} \lambda w\,.\,\mathcal{E}\llbracket(\mathcal{C}_d\llbracket t_1\rrbracket)\rrbracket\rho[v\mapsto w] \stackrel{\text{induction}}{=} \lambda w\,.\,\mathcal{S}_d\llbracket t_1\rrbracket\rho[v\mapsto w] \stackrel{\mathcal{S}_d}{=} \mathcal{S}_d\llbracket \mathit{Lam}\,v\,t_1\rrbracket\rho.$$

$$\mathcal{E}\llbracket\mathcal{C}_d\llbracket \mathit{App}\,t_1\;t_2\rrbracket\rrbracket\rho \stackrel{\mathcal{C}_d}{=} \mathcal{E}\llbracket \mathit{App}\,(\mathcal{C}_d\llbracket t_1\rrbracket)\,(\mathcal{C}_d\llbracket t_2\rrbracket)\rrbracket\rho \stackrel{\mathcal{E}}{=} (\mathcal{E}\llbracket\mathcal{C}_d\llbracket t_1\rrbracket\rrbracket\rho)\,(\mathcal{E}\llbracket\mathcal{C}_d\llbracket t_2\rrbracket\rrbracket\rho) \stackrel{\text{2 inductions}}{=} (\mathcal{S}_d\llbracket t_1\rrbracket\rho)\,(\mathcal{S}_d\llbracket t_2\rrbracket\rho) \stackrel{\mathcal{S}_d}{=} \mathcal{S}_d\llbracket \mathit{App}\,t_1\;t_2\rrbracket\rho.$$

$$\mathcal{E}\llbracket\mathcal{C}_d\llbracket \mathit{Let}\,v\,t_1\;t_2\rrbracket\rrbracket\rho \stackrel{\mathcal{C}_d}{=} \mathcal{E}\llbracket \mathit{Let}\,v\,(\mathcal{C}_d\llbracket t_1\rrbracket)\,(\mathcal{C}_d\llbracket t_2\rrbracket)\rrbracket\rho \stackrel{\mathcal{E}}{=} \mathcal{E}\llbracket\mathcal{C}_d\llbracket t_2\rrbracket\rrbracket\rho[v\mapsto\mathcal{E}\llbracket\mathcal{C}_d\llbracket t_1\rrbracket\rrbracket\rho] \stackrel{\text{2 inductions}}{=} \mathcal{S}_d\llbracket t_2\rrbracket\rho[v\mapsto\mathcal{S}_d\llbracket t_1\rrbracket\rho] \stackrel{\mathcal{S}_d}{=} \mathcal{S}_d\llbracket \mathit{Let}\,v\,t_1\;t_2\rrbracket\rho.$$

$$\begin{aligned}
&\mathcal{E}\llbracket\mathcal{C}_d\llbracket \underline{\mathit{Lam}}\,v\,t_1\rrbracket\rrbracket\rho \stackrel{\mathcal{C}_d}{=} \mathcal{E}\llbracket \mathit{Let}\,m\,\mathit{Fresh}\,(\mathit{Let}\,v\,(\mathit{Var}\diamond\,m)\,(\mathit{Lam}\diamond\,(\mathit{Var}\,m)\,(\mathcal{C}_d\llbracket t_1\rrbracket)))\rrbracket\rho \stackrel{\mathcal{E}}{=}\\
&\quad \mathcal{E}\llbracket \mathit{Let}\,v\,(\mathit{Var}\diamond\,m)\,(\mathit{Lam}\diamond\,(\mathit{Var}\,m)\,(\mathcal{C}_d\llbracket t_1\rrbracket))\rrbracket\rho[m\mapsto \mathit{fresh}()] \stackrel{\beta}{=}\\
&\quad \mathit{let}\;n{=}\mathit{fresh}()\;\mathit{in}\;\mathcal{E}\llbracket \mathit{Let}\,v\,(\mathit{Var}\diamond\,m)\,(\mathit{Lam}\diamond\,(\mathit{Var}\,m)\,(\mathcal{C}_d\llbracket t_1\rrbracket))\rrbracket\rho[m\mapsto n] \stackrel{\mathcal{E}}{=}\\
&\quad \mathit{let}\;n{=}\mathit{fresh}()\;\mathit{in}\;\mathcal{E}\llbracket \mathit{Lam}\diamond\,(\mathit{Var}\,m)\,\mathcal{C}_d\llbracket t_1\rrbracket\rrbracket\rho[m\mapsto n,\,v\mapsto\mathcal{E}\llbracket\mathit{Var}\diamond\,m\rrbracket\rho[m\mapsto n]] \stackrel{\mathcal{E};\,\mathcal{E};\,\mathcal{E}}{=}\\
&\quad \mathit{let}\;n{=}\mathit{fresh}()\;\mathit{in}\;\mathit{Lam}\,n\,(\mathcal{E}\llbracket\mathcal{C}_d\llbracket t_1\rrbracket\rrbracket\rho[m\mapsto n,\,v\mapsto\,\mathit{Var}\,n]) \stackrel{m\text{ not free in }\mathcal{C}_d\llbracket t_1\rrbracket}{=}\\
&\quad \mathit{let}\;n{=}\mathit{fresh}()\;\mathit{in}\;\mathit{Lam}\,n\,(\mathcal{E}\llbracket\mathcal{C}_d\llbracket t_1\rrbracket\rrbracket\rho[v\mapsto\,\mathit{Var}\,n]) \stackrel{\text{induction}}{=}\\
&\quad \mathit{let}\;n{=}\mathit{fresh}()\;\mathit{in}\;\mathit{Lam}\,n\,(\mathcal{S}_d\llbracket t_1\rrbracket\rho[v\mapsto\,\mathit{Var}\,n]) \stackrel{\mathcal{S}_d}{=} \mathcal{S}_d\llbracket\underline{\mathit{Lam}}\,v\,t_1\rrbracket\rho.
\end{aligned}$$

$$\mathcal{E}\llbracket\mathcal{C}_d\llbracket \underline{\mathit{App}}\,t_1\;t_2\rrbracket\rrbracket\rho \stackrel{\mathcal{C}_d}{=} \mathcal{E}\llbracket \mathit{App}\diamond\,(\mathcal{C}_d\llbracket t_1\rrbracket)\,(\mathcal{C}_d\llbracket t_2\rrbracket)\rrbracket\rho \stackrel{\mathcal{E}}{=} \mathit{App}\,(\mathcal{E}\llbracket\mathcal{C}_d\llbracket t_1\rrbracket\rrbracket\rho)\,(\mathcal{E}\llbracket\mathcal{C}_d\llbracket t_2\rrbracket\rrbracket\rho) \stackrel{\text{2 inductions}}{=} \mathit{App}\,(\mathcal{S}_d\llbracket t_1\rrbracket\rho)\,(\mathcal{S}_d\llbracket t_2\rrbracket\rho) \stackrel{\mathcal{S}_d}{=} \mathcal{S}_d\llbracket\underline{\mathit{App}}\,t_1\;t_2\rrbracket\rho.$$

$$\begin{aligned}
&\mathcal{E}\llbracket\mathcal{C}_d\llbracket \underline{\mathit{Let}}\,v\,t_1\;t_2\rrbracket\rrbracket\rho \stackrel{\mathcal{C}_d}{=} \mathcal{E}\llbracket \mathit{Let}\,m\,\mathit{Fresh}\,(\mathit{Let}\,v\,(\mathit{Var}\diamond\,m)\,(\mathit{Let}\diamond\,(\mathit{Var}\,m)\,(\mathcal{C}_d\llbracket t_1\rrbracket)\,(\mathcal{C}_d\llbracket t_2\rrbracket)))\rrbracket\rho \stackrel{\mathcal{E}}{=}\\
&\quad \mathcal{E}\llbracket \mathit{Let}\,v\,(\mathit{Var}\diamond\,m)\,(\mathit{Let}\diamond\,(\mathit{Var}\,m)\,(\mathcal{C}_d\llbracket t_1\rrbracket)\,(\mathcal{C}_d\llbracket t_2\rrbracket))\rrbracket\rho[m\mapsto \mathit{fresh}()] \stackrel{\beta}{=}\\
&\quad \mathit{let}\;n{=}\mathit{fresh}()\;\mathit{in}\;\mathcal{E}\llbracket \mathit{Let}\,v\,(\mathit{Var}\diamond\,m)\,(\mathit{Let}\diamond\,(\mathit{Var}\,m)\,(\mathcal{C}_d\llbracket t_1\rrbracket)\,(\mathcal{C}_d\llbracket t_2\rrbracket))\rrbracket\rho[m\mapsto n] \stackrel{\mathcal{E}}{=}\\
&\quad \mathit{let}\;n{=}\mathit{fresh}()\;\mathit{in}\;\mathcal{E}\llbracket \mathit{Let}\diamond\,(\mathit{Var}\,m)\,(\mathcal{C}_d\llbracket t_1\rrbracket)\,(\mathcal{C}_d\llbracket t_2\rrbracket)\rrbracket\rho[m\mapsto n,\,v\mapsto\mathcal{E}\llbracket\mathit{Var}\diamond\,m\rrbracket\rho[m\mapsto n]] \stackrel{\mathcal{E};\,\mathcal{E};\,\mathcal{E}}{=}\\
&\quad \mathit{let}\;n{=}\mathit{fresh}()\;\mathit{in}\;\mathit{Let}\,n\,(\mathcal{E}\llbracket\mathcal{C}_d\llbracket t_1\rrbracket\rrbracket\rho[m\mapsto n,\,v\mapsto\,\mathit{Var}\,n])\,(\mathcal{E}\llbracket\mathcal{C}_d\llbracket t_2\rrbracket\rrbracket\rho[m\mapsto n,\,v\mapsto\,\mathit{Var}\,n]) \stackrel{m\text{ not free in }\mathcal{C}_d\llbracket t_1\rrbracket,\mathcal{C}_d\llbracket t_2\rrbracket}{=}\\
&\quad \mathit{let}\;n{=}\mathit{fresh}()\;\mathit{in}\;\mathit{Let}\,n\,(\mathcal{E}\llbracket\mathcal{C}_d\llbracket t_1\rrbracket\rrbracket\rho[v\mapsto\,\mathit{Var}\,n])\,(\mathcal{E}\llbracket\mathcal{C}_d\llbracket t_2\rrbracket\rrbracket\rho[v\mapsto\,\mathit{Var}\,n]) \stackrel{\text{2 inductions}}{=}\\
&\quad \mathit{let}\;n{=}\mathit{fresh}()\;\mathit{in}\;\mathit{Let}\,n\,(\mathcal{S}_d\llbracket t_1\rrbracket\rho[v\mapsto\,\mathit{Var}\,n])\,(\mathcal{S}_d\llbracket t_2\rrbracket\rho[v\mapsto\,\mathit{Var}\,n]) \stackrel{v\text{ not free in }t_1\;(\alpha\text{-conv.})}{=}\\
&\quad \mathit{let}\;n{=}\mathit{fresh}()\;\mathit{in}\;\mathit{Let}\,n\,(\mathcal{S}_d\llbracket t_1\rrbracket\rho)\,(\mathcal{S}_d\llbracket t_2\rrbracket\rho[v\mapsto\,\mathit{Var}\,n]) \stackrel{\mathcal{S}_d}{=} \mathcal{S}_d\llbracket\underline{\mathit{Let}}\,v\,t_1\;t_2\rrbracket\rho.
\end{aligned}$$

Figure 9: Correctness of ds-*cogen*

It now follows that lhs and rhs have identical termination properties (since e is always evaluated in e_1) and that e_1 and e_2 are identical, except at those leaves where e_1 contains e and e_2 contains the value of m (we shall be sloppy and just write m below). To prove lhs = rhs, we then just have to consider the differing leaves, i.e. we have to prove $\mathcal{E}\llbracket e\rrbracket\rho[\ldots] = \mathcal{E}\llbracket(\mathit{Var}\,m)\rrbracket\rho[m\mapsto\mathcal{E}\llbracket e\rrbracket\rho,\ldots]$ where $\rho[\ldots]$ and $\rho[m\mapsto\mathcal{E}\llbracket e\rrbracket\rho,\ldots]$ are the environments that $\mathcal{E}$ will use when evaluating the $e/(\mathit{Var}\,m)$ leaves. But we know that $\mathcal{E}\llbracket(\mathit{Var}\,m)\rrbracket\rho[m\mapsto\mathcal{E}\llbracket e\rrbracket\rho,\ldots] = \mathcal{E}\llbracket e\rrbracket\rho$ since m was fresh and hence is not shadowed in $\kappa\,(\mathit{Var}\,m)$. We thus have to prove $\mathcal{E}\llbracket e\rrbracket\rho[\ldots] = \mathcal{E}\llbracket e\rrbracket\rho$ which holds if no free variables of e are shadowed (and rebound) in $\kappa\;e$.

But no κ ever shadows any variable: the only relevant continuations which potentially may shadow free variables are the continuations $\lambda x\,.\,\mathit{Let}\,v\,\mathit{Fresh}\,\ldots$ generated by $\mathcal{C}_{cp}$'s $\underline{\mathit{Let}}$-rule. However, since all source variable names are distinct and since κ is relevant and hence has been generated independently of t_1, variable x cannot possible become bound to any expression containing any (and hence no free) occurrences of variable v when computing $\mathcal{C}_{cp}\llbracket t_1\rrbracket(\lambda x.\ldots)$. □

LEMMA 6 (Reordering λ and *let*)

$$\forall\kappa : \lambda a\,.\,\mathit{let}\;m{=}\mathit{fresh}()\;\mathit{in}\;\mathcal{E}\llbracket\kappa\,(\mathit{Var}\,m)\rrbracket\rho[m\mapsto a] = \mathit{let}\;m{=}\mathit{fresh}()\;\mathit{in}\;\lambda a\,.\,\mathcal{E}\llbracket\kappa\,(\mathit{Var}\,m)\rrbracket\rho[m\mapsto a]$$

PROOF: Both sides of the equality terminate equally often. The difference between the two expressions is then only that the left-hand side generates a different m each time the function is applied whereas the right-hand side uses the same m. But as the value of $\mathcal{E}\llbracket\kappa\,(\mathit{Var}\,m)\rrbracket\rho[m\mapsto a]$ is independent of which particular fresh variable m denotes, the equality follows. □

LEMMA 7 (Reordering $\mathcal{E}$ and *let*)

$$\forall E_1, E_2 : n\text{ not free in }E_2 \;\Rightarrow\; \mathcal{E}\llbracket\mathit{let}\;n{=}\mathit{fresh}()\;\mathit{in}\;E_1\rrbracket E_2 = \mathit{let}\;n{=}\mathit{fresh}()\;\mathit{in}\;\mathcal{E}\llbracket E_1\rrbracket E_2$$

PROOF: Follows from strictness of $\mathcal{E}$ in its first argument and that the *let*-form is strict. The condition "n not free in E_2" ensures that no undesired shadowing occurs. □

LEMMA 8 (Removing superfluous fresh variable generation)
$\forall M : M$ not free in $E \;\Rightarrow\; \mathit{let}\, M{=}\mathit{fresh}()\;\mathit{in}\; E \;=\; E$

PROOF: Trivial as expression *fresh*() always terminates normally. □

Let us now give the inductive proof of Theorem 2. We use the textual abbreviation μ for the continuation $(\lambda a.\, \mathit{let}\, m{=}\mathit{fresh}()\;\mathit{in}\; \mathcal{E}[\![\kappa\,(\mathit{Var}\, m)]\!]\rho[m \;\mapsto\; a])$ that occurs in Theorem 2 and in Lemma 5. For each possible t, we thus have to prove $\mathcal{E}[\![\mathcal{C}_{cp}[\![t]\!]\kappa]\!]\rho = \mathcal{S}_{cp}[\![t]\!]\rho\mu$. Notice that, using the abbreviation, Lemma 5 states that $\mathcal{E}[\![\kappa\; E]\!]\rho = \mu\,(\mathcal{E}[\![E]\!]\rho)$.

For proof of theorem 2 see Figure 10 and Figure 11.

References

[BHOS76] Lennart Beckman, Anders Haraldson, Östen Oskarsson, and Erik Sandewall. A partial evaluator and its use as a programming tool. *Artificial Intelligence*, 7:319–357, 1976.

[Bon88] Anders Bondorf. Towards a self-applicable partial evaluator for term rewriting systems. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 27–50. North-Holland, 1988.

[Bon91] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17(1-3):3–34, December 1991. Revision of paper in ESOP'90, LNCS 432, May 1990.

[Bon92] Anders Bondorf. Improving binding times without explicit cps-conversion. In *1992 ACM Conference on Lisp and Functional Programming. San Francisco, California. LISP Pointers V, 1*, pages 1–10, June 1992.

[BW93] Lars Birkedal and Morten Welinder. Partial evaluation of Standard ML. Technical Report DIKU-report 93/22, DIKU, Department of Computer Science, University of Copenhagen, October 1993.

[DF90] Olivier Danvy and Andrzej Filinski. Abstracting control. In *1990 ACM Conference on Lisp and Functional Programming. Nice, France*, pages 151–160, June 1990.

[DF92] Olivier Danvy and Andrzej Filinski. Representing control. *Mathematical Structures in Computer Science*, 2(4), 1992.

[DNBV91] Anne De Niel, Eddy Bevers, and Karel De Vlaminck. Partial evaluation of polymorphically typed functional languages: the representation problem. In M. Billaud et al., editors, *Analyse Statique en Programmation Équationnelle, Fonctionnelle, et Logique, Bordeaux, France, Octobre 1991 (Bigre, vol. 74)*, pages 90–97. Rennes: IRISA, 1991.

[GJ91] Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.

[Gom90] Carsten K. Gomard. Partial type inference for untyped functional programs. In *1990 ACM Conference on Lisp and Functional Programming. Nice, France*, pages 282–287, June 1990.

[Hen91] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In John Hughes, editor, *Conference on Functional Programming and Computer Architecture, Cambridge, Massachusetts. Lecture Notes in Computer Science 523*, pages 448–472. Springer-Verlag, August 1991.

[HL91] Carsten Kehler Holst and John Launchbury. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming, Skye, Scotland*, pages 210–218. Glasgow University, 1991.

[Hol89] Carsten Kehler Holst. Syntactic currying: yet another approach to partial evaluation. Student Report 89-7-6, DIKU, University of Copenhagen, Denmark, July 1989.

[Lau91] John Launchbury. A strongly-typed self-applicable partial evaluator. In John Hughes, editor, *Conference on Functional Programming and Computer Architecture, Cambridge, Massachusetts. Lecture Notes in Computer Science 523*, pages 145–164. Springer-Verlag, August 1991.

[LD94] J. Lawall and O. Danvy. Continuation-based partial evaluation. In *1994 ACM Conference on Lisp and Functional Programming. Orlando, Florida*, June 1994.

[NN88] Hanne R. Nielson and Flemming Nielson. Automatic binding time analysis for a typed λ-calculus. In *Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. San Diego, California*, pages 98–106, 1988.

$$\mathcal{E}[\![\mathcal{C}_{cp}[\![\mathit{Var}\,v]\!]\kappa]\!]\rho \overset{\mathcal{C}_{cp}}{=} \mathcal{E}[\![\kappa\,(\mathit{Var}\,v)]\!]\rho \overset{\text{Lemma 5}}{=} \mu\,(\mathcal{E}[\![\mathit{Var}\,v]\!]\rho) \overset{\mathcal{E}}{=} \mu\,(\rho\,v) \overset{\mathcal{S}_{cp}}{=} \mathcal{S}_{cp}[\![\mathit{Var}\,v]\!]\rho\mu.$$

$$\begin{aligned}
&\mathcal{E}[\![\mathcal{C}_{cp}[\![\mathit{Lam}\,v\,t_1]\!]\kappa]\!]\rho \overset{\mathcal{C}_{cp}}{=} \mathcal{E}[\![\kappa\,(\mathit{Lam}\,v\,(\mathit{let}\,n{=}\mathit{fresh}()\ \mathit{in}\ \mathit{Lam}\,n\,(\mathcal{C}_{cp}[\![t_1]\!](\lambda x\,.\,\mathit{App}\,(\mathit{Var}\,n)\,x))))]\!]\rho \overset{\text{Lemma 5}}{=}\\
&\quad \mu\,(\mathcal{E}[\![\mathit{Lam}\,v\,(\mathit{let}\,n{=}\mathit{fresh}()\ \mathit{in}\ \mathit{Lam}\,n\,(\mathcal{C}_{cp}[\![t_1]\!](\lambda x\,.\,\mathit{App}\,(\mathit{Var}\,n)\,x)))]\!]\rho) \overset{\mathcal{E};\ \text{Lemma 7};\ \mathcal{E}}{=}\\
&\quad \mu\,(\lambda w\,.\,\mathit{let}\,n{=}\mathit{fresh}()\ \mathit{in}\ \lambda u\,.\,\mathcal{E}[\![\mathcal{C}_{cp}[\![t_1]\!](\lambda x\,.\,\mathit{App}\,(\mathit{Var}\,n)\,x)]\!]\rho[v\mapsto w,\,n\mapsto u]) \overset{\text{induction}}{=}\\
&\quad \mu\,(\lambda w\,.\,\mathit{let}\,n{=}\mathit{fresh}()\ \mathit{in}\ \lambda u\,.\,\mathcal{S}_{cp}[\![t_1]\!]\rho[v\mapsto w,\,n\mapsto u](\lambda a\,.\,\mathit{let}\,m{=}\mathit{fresh}()\ \mathit{in}\\
&\qquad \mathcal{E}[\![(\lambda x\,.\,\mathit{App}\,(\mathit{Var}\,n)\,x)\,(\mathit{Var}\,m)]\!]\rho[v\mapsto w,\,n\mapsto u,\,m\mapsto a])) \overset{n\text{ not free in }t_1;\ \beta;\ \mathcal{E}}{=}\\
&\quad \mu\,(\lambda w\,.\,\mathit{let}\,n{=}\mathit{fresh}()\ \mathit{in}\ \lambda u\,.\,\mathcal{S}_{cp}[\![t_1]\!]\rho[v\mapsto w](\lambda a\,.\,\mathit{let}\,m{=}\mathit{fresh}()\ \mathit{in}\ u\,a)) \overset{\text{Lemma 8 twice}}{=}\\
&\quad \mu\,(\lambda w\,.\,\lambda u\,.\,\mathcal{S}_{cp}[\![t_1]\!]\rho[v\mapsto w](\lambda a\,.\,u\,a)) \overset{\eta\text{ twice}}{=} \mu\,(\lambda w\,.\,\mathcal{S}_{cp}[\![t_1]\!]\rho[v\mapsto w]) \overset{\mathcal{S}_{cp}}{=} \mathcal{S}_{cp}[\![\mathit{Lam}\,v\,t_1]\!]\rho\mu.
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}[\![\mathcal{C}_{cp}[\![\mathit{App}\,t_1\ t_2]\!]\kappa]\!]\rho \overset{\mathcal{C}_{cp}}{=}\\
&\quad \mathcal{E}[\![\mathcal{C}_{cp}[\![t_1]\!](\lambda x\,.\,\mathcal{C}_{cp}[\![t_2]\!](\lambda y\,.\,\mathit{App}\,(\mathit{App}\,x\,y)\,(\mathit{let}\,n{=}\mathit{fresh}()\ \mathit{in}\ \mathit{Lam}\,n\,(\kappa\,(\mathit{Var}\,n)))))]\!]\rho \overset{\text{induction}}{=}\\
&\quad \mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda a\,.\,\mathit{let}\,m{=}\mathit{fresh}()\ \mathit{in}\ \mathcal{E}[\![(\lambda x\,.\,\mathcal{C}_{cp}[\![t_2]\!](\lambda y\,.\,\mathit{App}\,(\mathit{App}\,x\,y)\,(\mathit{let}\,n{=}\mathit{fresh}()\ \mathit{in}\\
&\qquad \mathit{Lam}\,n\,(\kappa\,(\mathit{Var}\,n)))))\,(\mathit{Var}\,m)]\!]\rho[m\mapsto a]) \overset{\beta;\ \text{renaming } a \text{ to } x}{=}\\
&\quad \mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda x\,.\,\mathit{let}\,m{=}\mathit{fresh}()\ \mathit{in}\ \mathcal{E}[\![\mathcal{C}_{cp}[\![t_2]\!](\lambda y\,.\,\mathit{App}\,(\mathit{App}\,(\mathit{Var}\,m)\,y)\,(\mathit{let}\,n{=}\mathit{fresh}()\ \mathit{in}\\
&\qquad \mathit{Lam}\,n\,(\kappa\,(\mathit{Var}\,n))))]\!]\rho[m\mapsto x]) \overset{\text{induction}}{=}\\
&\quad \mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda x\,.\,\mathit{let}\,m{=}\mathit{fresh}()\ \mathit{in}\ \mathcal{S}_{cp}[\![t_2]\!]\rho[m\mapsto x](\lambda a\,.\,\mathit{let}\,m'{=}\mathit{fresh}()\ \mathit{in}\ \mathcal{E}[\![(\lambda y\,.\,\mathit{App}\,(\mathit{App}\,(\mathit{Var}\,m)\,y)\\
&\qquad (\mathit{let}\,n{=}\mathit{fresh}()\ \mathit{in}\ \mathit{Lam}\,n\,(\kappa\,(\mathit{Var}\,n))))\,(\mathit{Var}\,m')]\!]\rho[m\mapsto x,\,m'\mapsto a])) \overset{m\text{ not free in }t_2;\ \beta;\ \mathcal{E};\ \text{Lemma 7};\ \mathcal{E}}{=}\\
&\quad \mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda x\,.\,\mathit{let}\,m{=}\mathit{fresh}()\ \mathit{in}\ \mathcal{S}_{cp}[\![t_2]\!]\rho(\lambda a\,.\,\mathit{let}\,m'{=}\mathit{fresh}()\ \mathit{in}\ (x\,a)\,(\mathit{let}\,n{=}\mathit{fresh}()\ \mathit{in}\ \lambda w\,.\,\mathcal{E}[\![\kappa\,(\mathit{Var}\,n)]\!]\\
&\qquad \rho[m\mapsto x,\,m'\mapsto a,\,n\mapsto w]))) \overset{m,\,m'\text{ not free in }\kappa\,(\mathit{Var}\,n);\ \text{Lemma 8 twice; renaming } a \text{ to } y,\ w \text{ to } a,\ n \text{ to } m}{=}\\
&\quad \mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda x\,.\,\mathcal{S}_{cp}[\![t_2]\!]\rho(\lambda y\,.\,(x\,y)\,(\mathit{let}\,m{=}\mathit{fresh}()\ \mathit{in}\ \lambda a\,.\,\mathcal{E}[\![\kappa\,(\mathit{Var}\,m)]\!]\rho[m\mapsto a]))) \overset{\text{Lemma 6}}{=}\\
&\quad \mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda x\,.\,\mathcal{S}_{cp}[\![t_2]\!]\rho(\lambda y\,.\,(x\,y)\,(\lambda a\,.\,\mathit{let}\,m{=}\mathit{fresh}()\ \mathit{in}\ \mathcal{E}[\![\kappa\,(\mathit{Var}\,m)]\!]\rho[m\mapsto a]))) \overset{\mathcal{S}_{cp}}{=} \mathcal{S}_{cp}[\![\mathit{App}\,t_1\ t_2]\!]\rho\mu.
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}[\![\mathcal{C}_{cp}[\![\mathit{Let}\,v\,t_1\ t_2]\!]\kappa]\!]\rho \overset{\mathcal{C}_{cp}}{=} \mathcal{E}[\![\mathcal{C}_{cp}[\![t_1]\!](\lambda x\,.\,\mathit{Let}\,v\,x\,(\mathcal{C}_{cp}[\![t_2]\!]\kappa))]\!]\rho \overset{\text{induction}}{=}\\
&\quad \mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda a\,.\,\mathit{let}\,m'{=}\mathit{fresh}()\ \mathit{in}\ \mathcal{E}[\![(\lambda x\,.\,\mathit{Let}\,v\,x\,\mathcal{C}_{cp}[\![t_2]\!]\kappa)\,(\mathit{Var}\,m')]\!]\rho[m'\mapsto a]) \overset{\beta;\ \mathcal{E};\ \text{renaming } a \text{ to } x}{=}\\
&\quad \mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda x\,.\,\mathit{let}\,m'{=}\mathit{fresh}()\ \mathit{in}\ \mathcal{E}[\![\mathcal{C}_{cp}[\![t_2]\!]\kappa]\!]\rho[m'\mapsto x,\,v\mapsto x]) \overset{\text{induction}}{=}\\
&\quad \mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda x\,.\,\mathit{let}\,m'{=}\mathit{fresh}()\ \mathit{in}\ \mathcal{S}_{cp}[\![t_2]\!]\rho[m'\mapsto x,\,v\mapsto x](\lambda a\,.\,\mathit{let}\,m{=}\mathit{fresh}()\ \mathit{in}\\
&\qquad \mathcal{E}[\![\kappa\,(\mathit{Var}\,m)]\!]\rho[m'\mapsto x,\,v\mapsto x,\,m\mapsto a])) \overset{m'\text{ not free in }t_2;\ m'\text{ not free in }\kappa\,(\mathit{Var}\,m);\ \text{Lemma 8}}{=}\\
&\quad \mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda x\,.\,\mathcal{S}_{cp}[\![t_2]\!]\rho[v\mapsto x](\lambda a\,.\,\mathit{let}\,m{=}\mathit{fresh}()\ \mathit{in}\ \mathcal{E}[\![\kappa\,(\mathit{Var}\,m)]\!]\rho[v\mapsto x,\,m\mapsto a])) \overset{\text{Lemma 4}}{=}\\
&\quad \mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda x\,.\,\mathcal{S}_{cp}[\![t_2]\!]\rho[v\mapsto x](\lambda a\,.\,\mathit{let}\,m{=}\mathit{fresh}()\ \mathit{in}\ \mathcal{E}[\![\kappa\,(\mathit{Var}\,m)]\!]\rho[m\mapsto a])) \overset{\mathcal{S}_{cp}}{=} \mathcal{S}_{cp}[\![\mathit{Let}\,v\,t_1\ t_2]\!]\rho\mu.
\end{aligned}$$

Figure 10: Correctness of cps-*cogen* (Part 1)

$$\mathcal{E}[\![\mathcal{C}_{cp}[\![\underline{Lam}\;v\;t_1]\!]\kappa]\!]\rho \overset{\mathcal{C}_{cp}}{=} \mathcal{E}[\![\kappa\,(Let\;m\;Fresh\,(Let\;v\,(Var\diamond\,m)\,(Lam\diamond\,(Var\;m)\,(\mathcal{C}_{cp}[\![t_1]\!]\iota))))]\!]\rho \overset{\text{Lemma 5}}{=}$$

$$\mu\,(\mathcal{E}[\![Let\;m\;Fresh\,(Let\;v\,(Var\diamond\,m)\,(Lam\diamond\,(Var\;m)\,(\mathcal{C}_{cp}[\![t_1]\!]\iota)))]\!]\rho) \overset{\mathcal{E};\;\underline{\beta};\;\mathcal{E}}{=}$$

$$\mu\,(let\;n{=}fresh()\;in\;\mathcal{E}[\![Lam\diamond\,(Var\;m)\,(\mathcal{C}_{cp}[\![t_1]\!]\iota)]\!]\rho[m\mapsto n,\;v\mapsto\mathcal{E}[\![Var\diamond\,m]\!]\rho[m\mapsto n]]) \overset{\mathcal{E};\;\underline{\mathcal{E}};\;\mathcal{E}}{=}$$

$$\mu\,(let\;n{=}fresh()\;in\;Lam\;n\,(\mathcal{E}[\![\mathcal{C}_{cp}[\![t_1]\!]\iota]\!]\rho[m\mapsto n,\;v\mapsto Var\;n])) \overset{m\text{ not free in }\mathcal{C}_{cp}[\![t_1]\!]\iota}{=}$$

$$\mu\,(let\;n{=}fresh()\;in\;Lam\;n\,(\mathcal{E}[\![\mathcal{C}_{cp}[\![t_1]\!]\iota]\!]\rho[v\mapsto Var\;n])) \overset{\text{induction}}{=}$$

$$\mu\,(let\;n{=}fresh()\;in\;Lam\;n\,(\mathcal{S}_{cp}[\![t_1]\!]\rho[v\mapsto Var\;n](\lambda a\,.\,let\;m{=}fresh()\;in\;\mathcal{E}[\![\iota\,(Var\;m)]\!]\rho[v\mapsto Var\;n,\;m\mapsto a])))$$

$$\overset{\mathcal{E};\text{ Lemma 8};\;\iota=\lambda a\,.\,a}{=} \mu\,(let\;n{=}fresh()\;in\;Lam\;n\,(\mathcal{S}_{cp}[\![t_1]\!]\rho[v\mapsto Var\;n]\iota)) \overset{\mathcal{S}_{cp}}{=} \mathcal{S}_{cp}[\![\underline{Lam}\;v\;t_1]\!]\rho\mu.$$

$$\mathcal{E}[\![\mathcal{C}_{cp}[\![\underline{App}\;t_1\;t_2]\!]\kappa]\!]\rho \overset{\mathcal{C}_{cp}}{=} \mathcal{E}[\![\mathcal{C}_{cp}[\![t_1]\!](\lambda x\,.\,\mathcal{C}_{cp}[\![t_2]\!](\lambda y\,.\,\kappa\,(App\diamond\,x\,y)))]\!]\rho \overset{\text{induction}}{=}$$

$$\mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda a\,.\,let\;m{=}fresh()\;in\;\mathcal{E}[\![(\lambda x\,.\,\mathcal{C}_{cp}[\![t_2]\!](\lambda y\,.\,\kappa\,(App\diamond\,x\,y)))\,(Var\;m)]\!]\rho[m\mapsto a]) \overset{\beta;\text{ renaming } a \text{ to } x}{=}$$

$$\mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda x\,.\,let\;m{=}fresh()\;in\;\mathcal{E}[\![\mathcal{C}_{cp}[\![t_2]\!](\lambda y\,.\,\kappa\,(App\diamond\,(Var\;m)\;y))]\!]\rho[m\mapsto x]) \overset{\text{induction}}{=}$$

$$\mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda x\,.\,let\;m{=}fresh()\;in\;\mathcal{S}_{cp}[\![t_2]\!]\rho[m\mapsto x](\lambda a\,.\,(let\;m'{=}fresh()\;in$$
$$\mathcal{E}[\![(\lambda y\,.\,\kappa\,(App\diamond\,(Var\;m)\;y))\,(Var\;m')]\!]\rho[m\mapsto x,\;m'\mapsto a]))) \overset{m\text{ not free in }t_2;\;\underline{\beta};\text{ renaming } a \text{ to } y}{=}$$

$$\mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda x\,.\,let\;m{=}fresh()\;in\;\mathcal{S}_{cp}[\![t_2]\!]\rho(\lambda y\,.\,(let\;m'{=}fresh()\;in$$
$$\mathcal{E}[\![\kappa\,(App\diamond\,(Var\;m)\,(Var\;m'))]\!]\rho[m\mapsto x,\;m'\mapsto y]))) \overset{\text{Lemma 5}}{=}$$

$$\mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda x\,.\,let\;m{=}fresh()\;in\;\mathcal{S}_{cp}[\![t_2]\!]\rho(\lambda y\,.\,(let\;m'{=}fresh()\;in$$
$$\mu\,(\mathcal{E}[\![App\diamond\,(Var\;m)\,(Var\;m')]\!]\rho[m\mapsto x,\;m'\mapsto y])))) \overset{\mathcal{E}}{=}$$

$$\mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda x\,.\,let\;m{=}fresh()\;in\;\mathcal{S}_{cp}[\![t_2]\!]\rho(\lambda y\,.\,(let\;m'{=}fresh()\;in\;\mu\,(App\;x\;y)))) \overset{\text{Lemma 8 twice}}{=}$$

$$\mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda x\,.\,\mathcal{S}_{cp}[\![t_2]\!]\rho(\lambda y\,.\,\mu\,(App\;x\;y))) \overset{\mathcal{S}_{cp}}{=} \mathcal{S}_{cp}[\![\underline{App}\;t_1\;t_2]\!]\rho\mu.$$

$$\mathcal{E}[\![\mathcal{C}_{cp}[\![\underline{Let}\;v\;t_1\;t_2]\!]\kappa]\!]\rho \overset{\mathcal{C}_{cp}}{=} \mathcal{E}[\![\mathcal{C}_{cp}[\![t_1]\!](\lambda x\,.\,Let\;m\;Fresh\,(Let\;v\,(Var\diamond\,m)\,(Let\diamond\,(Var\;m)\;x\,(\mathcal{C}_{cp}[\![t_2]\!]\kappa))))]\!]\rho \overset{\text{induction}}{=}$$

$$\mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda a\,.\,let\;m'{=}fresh()\;in\;\mathcal{E}[\![(\lambda x\,.\,Let\;m\;Fresh\,(Let\;v\,(Var\diamond\,m)\,(Let\diamond\,(Var\;m)\;x$$
$$(\mathcal{C}_{cp}[\![t_2]\!]\kappa))))\,(Var\;m')]\!]\rho[m'\mapsto a]) \overset{\beta;\;\mathcal{E};\;\beta;\;\mathcal{E};\;\mathcal{E};\;\mathcal{E};\;\mathcal{E};\;\mathcal{E};\text{ renaming } a \text{ to } x}{=}$$

$$\mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda x\,.\,let\;m'{=}fresh()\;in\;let\;n{=}fresh()\;in\;Let\;n\;x\,(\mathcal{E}[\![\mathcal{C}_{cp}[\![t_2]\!]\kappa]\!]\rho[m'\mapsto x,\;m\mapsto n,\;v\mapsto Var\;n]))$$

$$\overset{m\text{ not free in }\mathcal{C}_{cp}[\![t_2]\!]\kappa}{=} \mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda x\,.\,let\;m'{=}fresh()\;in\;let\;n{=}fresh()\;in\;Let\;n\;x\,(\mathcal{E}[\![\mathcal{C}_{cp}[\![t_2]\!]\kappa]\!]\rho[m'\mapsto x,\;v\mapsto Var\;n]))$$

$$\overset{\text{induction}}{=} \mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda x\,.\,let\;m'{=}fresh()\;in\;let\;n{=}fresh()\;in\;Let\;n\;x\,(\mathcal{S}_{cp}[\![t_2]\!]\rho[m'\mapsto x,\;v\mapsto Var\;n](\lambda a\,.\,(let\;m''{=}fresh()\;in$$
$$\mathcal{E}[\![\kappa\,(Var\;m'')]\!]\rho[m'\mapsto x,\;v\mapsto Var\;n,\;m''\mapsto a])))) \overset{m'\text{ not free in }t_2;\;m'\text{ not free in }\kappa\,(Var\;m'');\text{ Lemma 8}}{=}$$

$$\mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda x\,.\,let\;n{=}fresh()\;in\;Let\;n\;x\,(\mathcal{S}_{cp}[\![t_2]\!]\rho[v\mapsto Var\;n](\lambda a\,.\,(let\;m''{=}fresh()\;in$$
$$\mathcal{E}[\![\kappa\,(Var\;m'')]\!]\rho[v\mapsto Var\;n,\;m''\mapsto a])))) \overset{\text{Lemma 4}}{=}$$

$$\mathcal{S}_{cp}[\![t_1]\!]\rho(\lambda x\,.\,let\;n{=}fresh()\;in\;Let\;n\;x\,(\mathcal{S}_{cp}[\![t_2]\!]\rho[v\mapsto Var\;n](\lambda a\,.\,(let\;m''{=}fresh()\;in$$
$$\mathcal{E}[\![\kappa\,(Var\;m'')]\!]\rho[m\mapsto a])))) \overset{\mathcal{S}_{cp}}{=} \mathcal{S}_{cp}[\![\underline{Let}\;v\;t_1\;t_2]\!]\rho\mu.$$

Figure 11: Correctness of cps-*cogen* (Part 2)