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: \(cogen = mix(mix, mix)\), 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 \textit{continuation passing style} (abbreviated \textit{cps}) perform better than specializers written in direct style (abbreviated \textit{ds}) [Bon92]. For example, a specializer written in \textit{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 \textit{cps}-based specialization. We develop a handwritten \textit{cps}-\textit{cogen} which is superior to a \textit{ds}-\textit{cogen} for the same reason that a \textit{cps}-specializer is superior to a \textit{ds}-specializer: the \textit{cps}-\textit{cogen} can for example handle non-unfoldable let-expressions with static body. Hand-writing a \textit{cps}-\textit{cogen} is done along the same lines as hand-writing a \textit{ds}-\textit{cogen}, but some additional non-standard two-level \(\eta\)-expansions turn out to be needed.

The handwritten \textit{cps}-\textit{cogen} presented here is efficient in that it performs continuation processing (\(\beta\)-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 \textit{cogen} also contain continuations.

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

1 Introduction

Cps-based specializers are more powerful than \textit{ds}-based specializers. For example, a \textit{cps}-specializer straightforwardly specializes \((\text{let } y = \ldots \text{ in } \lambda x.x+y)\) into \((\text{let } y = \ldots \text{ in } 14+y)\) when the let-expression is non-unfoldable. The \textit{cps}-specializer is able to do so because it explicitly manipulates a context: a \textit{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 \textit{cps}-based \textit{cogen}. We derive the handwritten \textit{cps}-\textit{cogen} from a (handwritten) \textit{cps}-specializer. However, to make it easier to follow the derivation, we first show how to derive (and prove correctness of) a handwritten \textit{ds}-\textit{cogen} \(C_D\) from a (handwritten) \textit{ds}-specializer \(S_D\): the \textit{ds}-based \textit{cogen} is much simpler to derive than the \textit{cps}-based \textit{cogen}. Then we derive and prove correctness of the handwritten \textit{cps}-\textit{cogen} \(C_{cp}\) from a (handwritten) \textit{cps}-specializer \(S_{cp}\). See the horizontal arrows in Figure 1.

\begin{center}
\begin{tabular}{c c c}
\text{ds-specializer } & \longrightarrow & \text{ds-cogen } C_D \\
\downarrow & & \downarrow \\
\text{cps-specializer } S_{cp} & \longrightarrow & \text{cps-cogen } C_{cp}
\end{tabular}
\end{center}

\textbf{Figure 1: Overview}

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

We shall consider specialization similar to the one of \textit{lambda-mix} [GJ91]. In this paper we only consider a source language consisting of the \textit{strict} (call-by-value) weak-head normal form pure lambda calculus (variables, \(\lambda\)-abstraction and application) extended with a let-construct, see Figure 2.
We include the let-construct in the source language to cover a form that cps-based specialization treats better than "specialization does [Bon92].

\[
\begin{align*}
\text{Variable} &= \text{String} ; \quad e \in \text{Expression} ; \quad v \in \text{Variable} \\
\text{e} &:= \text{Var} \; v \mid \text{Lam} \; v \; e_1 \mid \text{App} \; e_1 \; e_2 \mid \text{Let} \; v \; e_1 \; e_2
\end{align*}
\]

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 \text{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 specialist 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 \text{Var}, \text{Lam}, \text{App} and \text{Let}. These additional constructors are \text{Var}, \text{Fresh}, \text{Lam}, \text{App} and \text{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. Notice that \text{fresh(v)} generates a fresh variable name (a string) and that the forms \text{Lam}, \text{App} and \text{Let} are used to generate expressions rather than values as \text{Lam}, \text{App} and \text{Let} do.

\[
\begin{align*}
\mathcal{E} : \text{Expression} \times (\text{Variable} \rightarrow \text{Value}) &\rightarrow \text{Value} \\
\mathcal{E}[\text{Var} \; v] \rho &= \rho \; v \\
\mathcal{E}[\text{Lam} \; v \; e] \rho &= \lambda w. \mathcal{E}[e] \rho[v[v \mapsto w]] \\
\mathcal{E}[\text{App} \; e_1 \; e_2] \rho &= (\mathcal{E}[e_1] \rho)(\mathcal{E}[e_2] \rho) \\
\mathcal{E}[\text{Let} \; v \; e_1 \; e_2] \rho &= \mathcal{E}[e_2] \rho[v \mapsto \mathcal{E}[e_1] \rho] \\
\mathcal{E}[\text{Var} \; v] \rho &= \text{Var} (\rho \; v) \\
\mathcal{E}[\text{Fresh}] \rho &= \text{fresh()} \\
\mathcal{E}[\text{Lam} \; v \; e] \rho &= \text{Lam}(\mathcal{E}[e] \rho)(\mathcal{E}[e] \rho) \\
\mathcal{E}[\text{App} \; e_1 \; e_2] \rho &= \text{App}(\mathcal{E}[e_1] \rho)(\mathcal{E}[e_2] \rho) \\
\mathcal{E}[\text{Let} \; v \; e_1 \; e_2] \rho &= \text{Let}(\mathcal{E}[e_1] \rho)(\mathcal{E}[e_2] \rho)(\mathcal{E}[e_2] \rho)
\end{align*}
\]

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. \text{Lam} \; v \; t.) and a dynamic version (e.g. \text{Lam} \; v \; t.) (for the dynamic version, 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 cogen-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 in [GJ91] (not for the let-form, though but it is simple to add). Annotating programs can be done automatically by \text{binding-time analysis}, see e.g. [Gom90, Hen91].

2 Direct style

Figure 5 specifies the ds-specializer \(S_d\). Specializer \(S_d\) 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 \(\mathcal{E}\) is a function \(\mathcal{E}\) in \text{Value}, since \text{Value} already includes the forms generated when evaluating the forms \text{Lam}, \text{App} and \text{Let} (Figure 3).

\[
\begin{align*}
\mathcal{E} : \text{Expression} \times (\text{Variable} \rightarrow \text{Value}) &\rightarrow \text{Value} \\
\mathcal{E}[\text{Var} \; v] \rho &= \rho \; v \\
\mathcal{E}[\text{Lam} \; v \; t.] \rho &= \lambda w. \mathcal{E}[t.] [\rho[v[v \mapsto w]] \\
\mathcal{E}[\text{App} \; t_1 \; t_2] \rho &= (\mathcal{E}[t_1] \rho)(\mathcal{E}[t_2] \rho) \\
\mathcal{E}[\text{Let} \; v \; t_1 \; t_2] \rho &= \mathcal{E}[t_2] \rho[v \mapsto \mathcal{E}[t_1] \rho] \\
\mathcal{E}[\text{Lam} \; v \; t.] \rho &= \text{let} \; v = \text{fresh()} \\
&\quad \text{in} \; \text{Lam} \; \mathcal{E}[t.] [\rho[v \mapsto \text{value}]] \\
\mathcal{E}[\text{App} \; t_1 \; t_2] \rho &= \text{App} \; (\mathcal{E}[t_1] \rho)(\mathcal{E}[t_2] \rho) \\
\mathcal{E}[\text{Let} \; v \; t_1 \; t_2] \rho &= \text{let} \; v = \text{fresh()} \\
&\quad \text{in} \; \text{Lam} \; \mathcal{E}[t_1] [\rho[v \mapsto \text{value}]] (\mathcal{E}[t_2] \rho[v \mapsto \text{value}])
\end{align*}
\]

Figure 5: Ds-specializer

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

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

Notice that \(C_d\) takes no environment \(\rho\) argument. Avoiding environment manipulation is possible by reusing source variable names in the treatments of \text{Lam}, \text{Let}, \text{Lam}
and \textit{Let} (notice e.g. how \textit{S}_d's \textit{Lam}-rule \(\lambda w. S_d [t] \rho[v \mapsto w]\) turns into \(\textit{Lam} v (C_d [t] \rho[v \mapsto w])\) in \textit{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 \textit{C}_d performs no symbolic unfolding and thus preserves the scope 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 co-gen \(C_d\) is indeed correct with respect to the specialization \(S_d\) (and in particular this also proves that the environment-free treatment of variables in \(C_d\) is correct). The theorem states that evaluating the code generated by \(C_d\) in environment \(\rho\) yields the same result as specializing by \(S_d\) (in environment \(\rho\)):

**Theorem 1 (Correctness of ds-cogen)**

\[ \forall t, \rho: \mathcal{C}_d [t] \mathcal{C}_d \rho = S_d [t] \rho \]

\textbf{Proof:} By structural induction over two-level expressions. See Appendix A.1 for details. \(\square\)

### 3 Continuation passing style

Figure 7 contains a \textit{cps}-specializer \(S_{cp}\), derived from \(S_d\) by (non-standard) \textit{cps}-transformation as described in [Bon92]; \textit{let} is the identity continuation \(\lambda z. z\). The \textit{cps}-specializer \(S_{cp}\) is more powerful than the \textit{ds}-specializer \(S_d\): it does not constrain the annotations of the body of \textit{Let}-forms (the type rule for checking well-annotatedness for \textit{Let}-forms is consequently more liberal for \textit{cps}-based specialization than for \textit{ds}-specialization). For example, the \textit{cp}-specializer \(S_{cp}\) can be used to specialize the form \(\textit{App} (\textit{Let} v \ldots (\textit{Lam} m \ldots)) (\textit{Var} m)\), hence \(\beta\)-reducing the application during specialization (contrasting to \(S_d\), cf. Section 2).

Notice that the identity continuation \(\lambda z. z\) is used not only to initialize, but also when treating \textit{Lam}-forms. This non-standard "impure" form of \textit{cps} turns out to be necessary to allow the desired liberal treatment of \textit{Let}-forms, propagating \(\kappa\) "over the let-binding". The more pure \textit{cps}-code \(\textit{let} n \leftarrow \textit{fresh} (1)\) in \(S_{cp} [t] \rho[v \mapsto n] (\lambda x. \kappa (\textit{Lam} n x))\) that one might have expected in the \textit{Lam}-rule thus gives an incorrect result if the lambda-body \(t\) is a \textit{Let}-form. Indeed, the let- and \(\lambda\)-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 \textit{Let}-forms; the continuation \(\lambda x. \kappa (\textit{Lam} m x)\) 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 \textit{cps-cogen} \(C_{cp}\), see Figure 8. Compiler generator \(C_{cp}\) is derived in the same way from \(S_{cp}\) as \(C_d\) was derived from \(S_d\) instead of performing what \(S_d\) does, \(C_{cp}\) generates code that will perform the same operations when evaluated. Deriving the \(C_{cp}\)-rules for \textit{Lam} and \textit{App} involves some additional steps that have no analogue in the \textit{C}_d-derivation; these steps will be described below. Notice that similarly to \(C_d\), compiler generator \(C_{cp}\) performs no operations on environments, contrasting to what a compiler generator generated by self-application would do. Also notice that \(C_{cp}\) has a continuation argument: we want \(C_{cp}\) to perform continuation reductions already at \textit{cogen}-time rather than suspending all continuation processing to appear in the programs generated by \textit{cogen} (such a simpler \textit{cps-cogen} can be written, but it is certainly less interesting).

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

\[ C_{cp} [\textit{Lam} v t] \kappa = \kappa (\textit{Lam} v (C_{cp} [t] \rho)) \]
\[ C_{cp} [\textit{App} t] \rho = C_{cp} [t] \rho (\lambda x. C_{cp} [t] \rho (\lambda y. \textit{App} (\textit{App} x y) \kappa)) \]

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

In a similar way, the \textit{App}-rule of Figure 8 is obtained from the incorrect one by \(\eta\)-expanding \(\kappa\) in the incorrect expression \(\textit{App} (\textit{App} x y) \kappa\) into \(\textit{Lam} n (\kappa (\textit{Var} n))\); \textit{App}'s second argument must be an expression, not a function.
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 Lam n (k (Var n)) in the App rule generates continuations that are present in the programs generated by C_{cp}. Thus, even though C_{cp} performs continuation processing (β-reductions), it also generates code that still contains (some) continuation processing. This is again analogous 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 C_{cp} with respect to S_{cp}, we must prove the following: for all t and ρ, it holds that \[\mathcal{E}[C_{cp}[t]] \rho = S_{cp}[t] \rho.\] That is, evaluating the expression generated by C_{cp} in some environment gives the same result as specializing t in the same environment. Both C_{cp} and S_{cp} are initially called with the identity continuation \(\lambda\).

To prove this equality inductively, we need a more general theorem that holds not only when the continuations are \(\lambda\). Can we hope to simply replace \(\lambda\) by \(\kappa\) and then expect that the equality holds for all \(\kappa\)? The answer is unfortunately "no".

The reason is simple: the type of \(S_{cp}\)'s continuation parameter is \(\text{Value} \rightarrow \text{Value}\) whereas the type of \(C_{cp}\)'s continuation parameter is \(\text{Expression} \rightarrow \text{Expression}\). However, given a \(C_{cp}\)-type continuation \(\kappa\), we can construct a \(S_{cp}\)-type continuation: \(\lambda a. \text{let } m = \text{fresh}() \in \mathcal{E}[\kappa (\text{Var } m)] (\rho [m \mapsto a].\) The idea here is to evaluate the expression generated by applying \(\kappa\) to an argument, taking care not to evaluate \(a\) which already is a \(\text{Value}\) (this is the reason why the continuation is not simply \(\lambda a. \mathcal{E} [\kappa a] \rho\)). This leads to the following correctness theorem.

\textbf{THEOREM 2} (Correctness of cps-cogen)
\[\forall t, \rho, \kappa : \mathcal{E}[C_{cp}[t]] \rho = S_{cp}[t] \rho.\]
\textbf{PROOF:} By structural induction over two-level expressions. See Appendix A.2 for details.

In this theorem, as well as in Appendix A.2, we implicitly assume some restrictions on \(\kappa\) when quantifying by \(\forall \kappa, \ldots, \kappa: \ldots\): continuation \(\kappa\) must be related to two-level expression \(t\) in the sense that \(\kappa\) only ranges over those continuations that are generated when computing \(C_{cp}[t]: \) where \(t\) is a subexpression of \(t_0\). That is, we only consider the relevant continuations, not all continuations. Notice that the identity continuation \(\lambda\) is a relevant continuation (possible value for \(\kappa\)).

The desired correctness property now follows as a corollary:

\textbf{COROLLARY 3} (Correctness of cps-cogen)
\[\forall t, \rho : \mathcal{E}[C_{cp}[t]] \rho = S_{cp}[t] \rho.\]
\textbf{PROOF:} Follows from Theorem 2 since
\[\lambda a. \text{let } m = \text{fresh}() \in \mathcal{E}[\kappa (\text{Var } m)] (\rho [m \mapsto a].\]
\[\lambda a. \text{let } m = \text{fresh}() \in \mathcal{E}[\kappa (\text{Var } m)] (\rho [m \mapsto a].\]
\[\lambda a. \text{let } m = \text{fresh}() \in \mathcal{E}[\kappa (\text{Var } m)] (\rho [m \mapsto a].\]

(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 \(\kappa\) are restricted as described earlier: all variable names in \(t\) must be distinct (\(\alpha\)-conversion, cf. Section 1), and \(\kappa\) must be relevant.
4 Deriving \( C_{cp} \) from \( C_d \)

In retrospect, when comparing \( C_d \) and \( C_{cp} \), we notice that \( C_{cp} \) could have been derived from \( C_d \) rather than from \( S_{cp} \); by \( \text{cps}-\)transforming the \( C_d \), taking into account to use the non-standard \( \text{cps} \) \( \text{Lam} \)-rule, and performing appropriate \( \eta \)-expansions for the \( \text{Lam} \)- and \( \text{App} \)-rules. This way of deriving \( S_{cp} \) might be useful in a context where a handwritten \( \text{ds}-\text{cogen} \) already exists, for example if one were to write a \( \text{cps}-\text{cogen} \) for the ML-\( \text{cogen} \) described in [BW03]. We believe that this can be done without great difficulty.

5 Related work

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

In [Hol89], a handwritten \( \text{cogen} \) was based on macro expansion. In the paper [HL91], a \( \text{ds}-\text{cogen} \) for a statically typed language is described. The ideas from [HL91] were used for hand-writing a \( \text{ds}-\text{cogen} \) 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 \( \text{cps} \)-specializer from [Bon92] can be almost automatically derived from a \( \text{ds} \)-specializer by inserting the control operators shift and reset (see [DF90]) at selected places and \( \text{cps} \) converting the resulting specialiser. They also devote some attention to how their ideas could be used in the context of a handwritten \( \text{cogen} \).

6 Conclusion

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

We believe that our handwritten \( \text{cogen} \) is a good starting point for hand-writing \( \text{cps} \)-based \( \text{cogens} \) for larger strict functional languages. Our work does not immediately carry over to lazy languages as the \( \text{cps} \)-transformation we have used is the strict \( \text{cps} \)-transformation. However, it is plausible that a similar development could be made for a lazy language using call-by-name \( \text{cps} \)-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 \( \beta \)- and \( \eta \)-equalities are used; \( \beta/\eta \) do not in general hold for the typed (\( C_d \) and \( S_d \) are both simply typed) strict weak-head normal form lambda-calculus. \( \beta/\eta \) thus only hold when termination properties do not change; we only use \( \beta/\eta \) when this is the case. We use \( \beta \)-abstraction to prevent duplicating expressions of form \( \text{fresh}() \). Also notice that in both proofs we rely on the fact that the variable \( m \), introduced in the \( \text{Lam} \)- and \( \text{Let} \)-rule in both \( \text{ds} \)- and \( \text{cps} \)-\( \text{cogen} \), is unique: \( m \) does not occur in input programs and cannot be generated by application of \( \text{fresh}() \). By construction it is assured that \( \forall t:_\{ \text{Lam} \}[t] \) (where \( \kappa \) is relevant) nor \( C_d[t] \) 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 \( t \) that ranges over object expressions). Recall (Section 3) that only two-level expressions \( t \) with all variable names distinct and only well-behaved continuations \( \kappa \) are considered when quantifying over \( t \) and \( \kappa \).

**Lemma 4** (Environment simplification)

\[ \forall t, \kappa : \text{if } v \text{ is bound in } t \text{ then} \]
\[ \forall \rho : \text{let } m = \text{fresh}() \text{ in } \mathcal{E}([\kappa][x](\text{Var } m)) \rho[v \mapsto \ldots] = \text{let } m = \text{fresh}() \text{ in } \mathcal{E}([\kappa][x](\text{Var } m)) \rho \]

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

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

**Lemma 5** (Extracting out \( \kappa \)'s argument)

\[ \forall t, \kappa : \text{if } t \text{ is one of the forms } \text{Var } \nu, \text{Lam } \nu \text{, } \text{Lam } \nu \text{, or } \text{App } \nu \text{, then when computing } C_{cp}[t][\kappa], \text{the following equality holds for (all relevant instances of)} \]
\[ \text{the expressions } \kappa E \text{ in the right-hand sides of the sides of the rules for } \text{Var} \text{, } \text{Lam} \text{, } \text{Lam} \text{, and } \text{App}: \]
\[ \forall \rho : \mathcal{E}([\kappa E]) \rho = (\lambda a. \text{let } m = \text{fresh}() \text{ in } \mathcal{E}([\kappa(\text{Var } m)]\rho[m \mapsto a])(\mathcal{E}[E]) \rho) \]

**Proof:** First notice that since \( \rho[m \mapsto a] \) is strict in \( a \), we may \( \beta \)-reduce \((\lambda a. \ldots)(\mathcal{E}[E]) \rho\). We thus have to prove \( \mathcal{E}([\kappa E]) \rho = \text{let } m = \text{fresh}() \text{ in } \mathcal{E}([\kappa(\text{Var } m)]\rho[m \mapsto a])(\mathcal{E}[E]) \rho \).

We shall refer to the left- and right-hand sides of this equality as \( \text{lhs} \) and \( \text{rhs} \) 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 \( \kappa(\text{Var } m) \); notice from the type of \( \kappa \) (Figure 8) that the values \( e_1 \) 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 \( \kappa \).
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 just have to consider the differing leaves, i.e. we have to prove $E[e_1] e_1 \ldots = E[e_2] e_2 \ldots$, where $e_1 \ldots$ and $e_2 \ldots$ are the environments that $E$ will use when evaluating the $e_i$'s ($Var m$).

We know that $E[e_1] e_1 \ldots = E[e_2] e_2 \ldots$ since $m$ was fresh and hence is not shadowed in $\kappa$ ($Var m$). We thus have to prove $E[e_1] \rho \ldots = E[e_2] \rho$ which holds if no free variables of $e$ are shadowed (and rebound) in $\kappa$.

But no $\kappa$ ever shadows any variable: the only relevant continuations which potentially may shadow free variables are the continuations $\lambda x. \text{let } v \text{ fresh } \ldots$ generated by $C_\text{rp}$'s $\text{Let}$-rule. However, since all source variable names are distinct and since $\kappa$ is relevant and hence has been generated independently of $t_1$ variable $x$ cannot possibly become bound to any expression containing any (and hence no free) occurrences of variable $v$ when computing $C_\text{rp}[t_1] (\lambda x \ldots)$. \hfill $\Box$.

**Lemma 6 (Reordering $\lambda$ and let)**

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

**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 $E^! \kappa (Var m) \rho [m \mapsto a]$ is independent of which particular fresh variable $m$ denotes, the equality follows. \hfill $\Box$.

**Lemma 7 (Reordering $E$ and let)**

$\forall E_1, E_2 : n \text{ not free in } E_2 \Rightarrow E^! \\text{let } n = \text{fresh } \in E_1 \square E_2 = \text{let } n = \text{fresh } \in E^! E_1 E_2$
Lemma 8 (Removing superfluous fresh variable generation)

∀ M : M not free in E ⇒ let M = fresh() 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 (λa. let m = fresh() in E[a (Var m)] m → a) that occurs in Theorem 2 and in Lemma 5. For each possible t, we thus have to prove E[μ|E]|ρ = Sμ|E|ρ. Notice that, using the abbreviation, Lemma 5 states that E|κ|E|ρ = μ(κ|E|ρ).

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

References


Figure 10: Correctness of cps-cogen (Part 1)
Figure 11: Correctness of cps-cogen (Part 2)