Cogen in Six Lines

Peter J. Thiemann*

Abstract

We have designed and implemented a program-generator generator (PGG) for an untyped higher-order functional programming language. The program generators perform continuation-based multi-level offline specialization and thus combine the most powerful and general offline partial evaluation techniques. The correctness of the PGG is ensured by deriving it from a multi-level specializer. Our PGG is extremely simple to implement due to the use of multi-level techniques and higher-order abstract syntax.

Keywords: partial evaluation, multi-level computation, continuations.

1 Introduction

An attractive feature of partial evaluation is the ability to generate *generating extensions*. A generating extension for a program `p` with two inputs $\texttt{inp}_s$ and $\texttt{inp}_d$ is a program `p-gen` which accepts the static input $\texttt{inp}_s$ of `p` and produces a *residual program* $\texttt{p}_s$ which accepts the dynamic input $\texttt{inp}_d$ and produces the same result as $[\![\texttt{p}]\!]\ \texttt{inp}_s\ \texttt{inp}_d$, provided both `p` and $\texttt{p}_s$ terminate.

$$
\begin{aligned}
&[\![\texttt{p-gen}]\!]\ \texttt{inp}_s = \texttt{p}_s\\
&[\![\texttt{p}_s]\!]\ \texttt{inp}_d = \texttt{result}\\
&[\![\texttt{p}]\!]\ \texttt{inp}_s\ \texttt{inp}_d = \texttt{result}
\end{aligned}
$$

Generating extensions result from applying a program-generator generator (PGG) to `p`. Customarily, PGGs result from double self-application of a partial evaluator as described by the third Futamura projection [15, 20]. They are often called compiler generators (*cogen*) or partial evaluation compilers (*pecom*) as they can turn interpreters into compilers. We will use *pecom* for a PGG constructed by self-application of a partial evaluator.

The interest in generating extensions derives not only from theoretical considerations. Using generating extensions speeds up specialization by a factor of three to four compared with a generic specializer [4]. This is also called the *cogen approach* to partial evaluation.

Writing PGGs by hand is a recent trend in partial evaluation [23, 3, 16]. This technique solves some data representation and efficiency problems:

- Partial evaluation for typed languages suffers from an encoding problem because static values and programs need to be represented as values [22, 1, 13]. As a consequence, self-application is impractical, and explicit tagging and untagging operations may remain in the residual programs.
- *Pecom* inherits some inefficiencies from the original specializer due to its interpretive nature.
- *Pecom* can only make use of language features which are dealt with by the specializer. This restriction also applies to the generating extensions.

Writing the PGG by hand makes all these problems disappear: the PGG only handles syntax trees of the language, it does not contain an interpreter, and all language features can be freely used inside the PGG as well as in the generating extensions.

An important feature of offline partial evaluators is *continuation-based specialization* [12, 5, 8, 24, 25]. It is a fully automatic binding-time improving transformation that facilitates avoiding duplication of computation, thereby simplifying unfolding strategies and the implementation of partially static data.

We develop multi-level continuation-based specialization and extend the approach to the construction of a hand-written multi-level continuation-based PGG. Contrary to the usual ad-hoc construction of PGGs we provide a simple derivation of the PGG from a specializer. A key feature in our derivation is the transformation to higher-order abstract syntax (HOAS) which considerably simplifies the resulting PGG and the proof of its correctness. By exploiting the multi-level features it is possible to write the PGG in direct style without using control operators. Consequently, multi-level specialization, which is interesting in its own right, mainly serves to simplify the construction of the PGG. As far as we know, multi-level specialization has not been combined with continuation-based partial evaluation to date.

Finally, we demonstrate that the techniques presented scale up to the functional subset of Scheme and exhibit remarkable speed-ups compared to an available state-of-the-art partial evaluator.

Overview In Sec. 2, we introduce some notation. Section 3 presents multi-level partial evaluation and explains the standard approach and the cogen approach. Continuation-based specialization is the subject of Sec. 4. Section 5 generalizes continuation-based reduction to more than two levels and contains the derivation of PGGs from specializers—all written in continuation-passing style. The following Section 6 performs the same development for direct style PGGs and specializers using control operators. We extend our approach to the full Scheme language in Sec. 7. Empirical data on our PGG including performance figures are presented in Sec. 8. In Sec. 9 we consider related work.

*Address: Wilhelm-Schickard-Institut, Universität Tübingen, Sand 13, D-72076 Tübingen, Germany. E-mail: thiemann@informatik.uni-tuebingen.de. In: ACM SIGPLAN International Conference on Functional Programming, ICFP'96, Philadelphia, USA.

2 Notational Preliminaries

The source and target language of our specializers consists of multi-level expressions E. For brevity's sake we restrict the formal presentation to the λ-mix fragment,

$$\begin{array}{lcl} E & \in & \mathit{MExpr} \\ E & ::= & v \mid \underline{\lambda}_i v.E \mid E \underline{@}_i E \mid \underline{\texttt{let}}_i\ v = E\ \underline{\texttt{in}}\ E \end{array}$$

where $0 \leq i < m$ and $m > 1$ is the number of levels. The set of free variables of E, $FV(E)$ is defined as usual.

The metalanguage used in all definitions is an enriched, untyped, call-by-value lambda calculus extended with quoting constructs.

$$\begin{array}{lcl} e & \in & \mathit{Expr} \\ e & ::= & v \mid o(e \ldots e) \mid \lambda v.e \mid (e\ e) \mid \texttt{let}\ v = e\ \texttt{in}\ e \mid \\ & & \lceil e \rceil \mid \langle e \rangle \end{array}$$

Expressions e are variables v, applications of built-in operators o (including constants), abstractions, applications, and `let` expressions. Additionally, quoted $\lceil e \rceil$ and unquoted expressions $\langle e \rangle$ correspond to Scheme's `quasiquote` and `unquote` forms [19].

In the metalanguage we deal with values of the following types.

- *Var*, variables,
- *Value*, everything which is not an *MExpr*,
- $\mathit{PEVal} = \mathit{MExpr} + \mathit{Value}$, specialization-time values,
- $\mathit{PEEnv} = \mathit{Var} \rightarrow \mathit{PEVal}$, specializer environments,
- $\mathit{PECont} = \mathit{PEVal} \rightarrow \mathit{PEVal}$, specializer continuations.

For $\rho \in \mathit{PEEnv}$, $\rho[v \mapsto e]$ denotes the usual extension of environment ρ by the binding $v \mapsto e$. For an expression $e \in \mathit{Expr}$, $e[v := e']$ denotes substitution of e' for v in e. Variables in lower case are usually of type *Value* wheras variables in upper case are of type *MExpr*. We let $=$ denote $\beta\eta$-equivalence of expressions, fresh variables introduced on right sides of definitions are marked with $^\diamond$.

Below, we consider the syntax constructors $\underline{\lambda}_i$, $\underline{@}_i$, and $\underline{\texttt{let}}_i$ as built-in operators of the metalanguage with types

$$\begin{array}{lcl} \underline{\lambda}_i & : & \mathit{Var} \times \mathit{MExpr} \rightarrow \mathit{MExpr} \\ \underline{@}_i & : & \mathit{MExpr} \times \mathit{MExpr} \rightarrow \mathit{MExpr} \\ \underline{\texttt{let}}_i & : & \mathit{Var} \times \mathit{MExpr} \times \mathit{MExpr} \rightarrow \mathit{MExpr} \end{array}$$

3 Multi-Level Partial Evaluation

Partial evaluation is a program specialization technique based on aggressive constant propagation [11, 20]. In multi-level partial evaluation [16], the execution of a program is staged in a finite number $m > 1$ of levels, guided by an initial assignment of binding times $0, \ldots, m-1$ to the inputs. The smaller the binding time, the earlier the value becomes available. We call values *static* if they are available at level 0, otherwise they are *dynamic*.

3.1 The Standard Approach

An offline partial evaluator consists of a binding-time analysis and a reducer. The binding-time analysis transforms a subject program to an annotated program marking every expression with the least level where it can be executed without referring to values that only become available later. The reducer processes the annotated program and the level-0 (static) parts of the input. It produces a residual program by reducing level-0 expressions and rebuilding dynamic ones with their level decremented by one. Unlike standard two-level partial evaluation, the residual program may still be an annotated program which can be passed to the reducer once more.

Let `bta` be the binding-time analysis, `red` the static reducer, `p` the subject program, $\texttt{inp}_i$ the input at level i, `result` the final result, b the initial binding-time assignment, and $m > 1$ the number of levels. Specialization of an m-level program works as follows:

$$\begin{array}{lcl} \texttt{p-ann}_0 & = & [\![\texttt{bta}]\!]\ \texttt{p}\ b \\ \texttt{p-ann}_1 & = & [\![\texttt{red}]\!]\ \texttt{p-ann}_0\ \texttt{inp}_0 \\ \ldots & & \\ \texttt{p-ann}_{m-1} & = & [\![\texttt{red}]\!]\ \texttt{p-ann}_{m-2}\ \texttt{inp}_{m-2} \\ \texttt{result} & = & [\![\texttt{p-ann}_{m-1}]\!]\ \texttt{inp}_{m-1} \end{array}$$

In the final step, $\texttt{p-ann}_{m-1}$ is a level-0 program and hence executable with the standard semantics. The mix equation—the correctness criterion for the specializer—generalizes to

$$\begin{array}{l} [\![\texttt{p}]\!]\ \texttt{inp}_0 \ldots \texttt{inp}_{m-1} = \\ \quad [\![[\![\texttt{red}]\!] \ldots ([\![\texttt{red}]\!]\ \texttt{p-ann}_0\ \texttt{inp}_0) \ldots\ \texttt{inp}_{m-2}]\!]\ \texttt{inp}_{m-1}. \end{array}$$

3.2 The Cogen Approach

As opposed to using a standard partial evaluator directly, constructing a generating extension first and performing the specialization proper with the generating extension results in three to four times faster specialization [4]. We can get the benefit of using generating extensions here by constructing a generating extension `red-gen` for `red` with a standard two-level *pecom*. From the mix equation and the third Futamura projection, we find

$$\texttt{p-ann}_{i+1} = [\![\texttt{red}]\!]\ \texttt{p-ann}_i\ \texttt{inp}_i = [\![[\![\texttt{red-gen}]\!]\ \texttt{p-ann}_i]\!]\ \texttt{inp}_i.$$

Using `red-gen`, the above specialization pipeline is shown in Fig. 1. However, the binding-time-annotated program `p-ann` is already a generating extension `p-gen` given a suitable interpretation of the annotations [17, 16]. This effectively removes every second step in the `red-gen` specialization pipeline. That is,

$$\texttt{p-ann}_{i+1} = [\![\texttt{p-ann}_i]\!]\ \texttt{inp}_i$$

This observation has been exploited by Glück and Jørgensen [16] who present a library of functions to interpret multi-level annotated programs as multi-level generating extensions. We extend their approach by integrating it with continuation-based partial evaluation [5, 24], once in continuation-passing style and once in direct style with control operators.

4 Continuation-Based Reduction

Context propagation is an important binding-time improvement. The standard example is

$$(\texttt{let}\ x = d\ \texttt{in}\ 17) + 4$$

$$
\begin{array}{lcllcl}
\texttt{p-ann}_0 & = & \llbracket\texttt{bta}\rrbracket\ \texttt{p}\ b & \texttt{p-gen}_0 & = & \llbracket\texttt{red-gen}\rrbracket\ \texttt{p-ann}_0 \\
\texttt{p-ann}_1 & = & \llbracket\texttt{p-gen}_0\rrbracket\ \texttt{inp}_0 & \texttt{p-gen}_1 & = & \llbracket\texttt{red-gen}\rrbracket\ \texttt{p-ann}_1 \\
\dots & & & & & \\
\texttt{p-ann}_{m-1} & = & \llbracket\texttt{p-gen}_{m-2}\rrbracket\ \texttt{inp}_{m-2} & \texttt{result} & = & \llbracket\texttt{p-ann}_{m-1}\rrbracket\ \texttt{inp}_{m-1}
\end{array}
$$

Figure 1: Multi-level Specialization Pipeline

with dynamic d. A naïve specializer regards the entire expression ($\texttt{let}\ x = d\ \texttt{in}\ 17$) as dynamic and does not reduce the above expression.

Continuation-based reduction [5, 24] propagates the static context $[\ ] + 4$ over the dynamic $\texttt{let}$ and the addition can be performed at specialization time. Now the residual code is

$$\texttt{let}\ x = d\ \texttt{in}\ 21.$$

Much of the importance of this kind of context propagation originates from the fact that dynamic $\texttt{let}$s are indispensable to avoid duplication of computation. Often, these $\texttt{let}$s are automatically inserted around function bodies and arguments of partially static data constructors by the specializer [7]. Therefore, continuation-based reduction is a vital feature for offline partial evaluators.

It must be noted, that the basic idea of this kind of context propagation is already present in Danvy and Filinski's work on a one-pass transformation to continuation-passing style (CPS) [12]. In fact, their CPS transformation is really a continuation-based reducer processing a hard-wired CPS interpreter. In other words, it is a generating extension performing continuation-based reduction.

As a running example, we will consider specializing the source expression

$$\underline{\lambda}_1 x.(\underline{\texttt{let}}_1\ v = a\underline{@}_1 b\ \underline{\texttt{in}}\ (\underline{\lambda}_0 z.z))\underline{@}_0 x$$

where a and b are dynamic variables. The static context $[\ ]\underline{@}_0 x$ must be propagated inside of the dynamic $\texttt{let}$ in order to facilitate any reduction.

5 Multi-level Continuation-Based Reduction

In order to generalize continuation-based reduction to more than two levels it is only necessary to distinguish between levels > 0 (dynamic) and level 0 (static). This is the only place where dynamic parts can corrupt static contexts. The reducer (see Fig. 2) is easily adapted from published accounts [5, 20]. The result from specializing the example is

$$
\begin{array}{r}
\mathcal{R}_c\llbracket\underline{\lambda}_1 x.(\underline{\texttt{let}}_1\ v = a\underline{@}_1 b\ \underline{\texttt{in}}\ (\underline{\lambda}_0 z.z))\underline{@}_0 x\rrbracket \\
[a \mapsto a, b \mapsto b](\lambda z.z) \\
= \underline{\lambda}_0 x.(\underline{\texttt{let}}_0\ v = a\underline{@}_0 b\ \underline{\texttt{in}}\ x).
\end{array}
$$

5.1 Introducing Higher-Order Abstract Syntax

We aim at interpreting a multi-level annotated program as a program generator [17, 16]. In order to find the interpretations of the syntax constructors $\underline{\lambda}_i$, $\underline{@}_i$, and $\underline{\texttt{let}}_i$ we derive their defining equations from the specializer shown in Fig. 2. In a first (trivially correct) transformation we simply transfer the abstraction of the continuation κ from the left side to the right side of the equations, as illustrated by

$$\mathcal{R}_c\llbracket v\rrbracket\rho = \lambda\kappa.\kappa(\rho\llbracket v\rrbracket).$$

Now we almost have a definition of the necessary operators. Only the environment ρ stands in the way of making it a stand-alone definition. Therefore, we change the way the abstract syntax is presented from first-order to higher-order.

Higher-order abstract syntax (HOAS) [28] is a way to transfer the burden to correctly handle bindings, scoping, substitution, and so on to the metalevel. Using HOAS, the binding occurrences of variables vanish and—after an intermediate step—we can use metavariables to represent variables. Using $MExpr'$ as type for a term in HOAS, we introduce the syntax constructors:

$$
\begin{array}{lcl}
\underline{\underline{\lambda}}_i & : & (MExpr' \to MExpr') \to MExpr' \\
\underline{\underline{@}}_i & : & MExpr' \times MExpr' \to MExpr' \\
\underline{\underline{\texttt{let}}}_i & : & MExpr' \times (MExpr' \to MExpr') \to MExpr'
\end{array}
$$

We can freely transform from HOAS and back (up to variable names) using straightforward functions $\Phi : MExpr \to MExpr'$ and $\Psi : MExpr' \to MExpr$.

$$
\begin{array}{lcl}
\Phi\llbracket v\rrbracket & = & \texttt{var}\ v \\
\Phi\llbracket\underline{\lambda}_i v.E\rrbracket & = & \underline{\underline{\lambda}}_i \lambda v.\Phi\llbracket E\rrbracket \\
\Phi\llbracket E_1\underline{@}_i E_2\rrbracket & = & \Phi\llbracket E_1\rrbracket\underline{\underline{@}}_i \Phi\llbracket E_2\rrbracket \\
\Phi\llbracket\underline{\texttt{let}}_i\ v = E_1\ \underline{\texttt{in}}\ E_2\rrbracket & = & \underline{\underline{\texttt{let}}}_i\ \Phi\llbracket E_1\rrbracket\ \underline{\underline{\texttt{in}}}\ \lambda v.\Phi\llbracket E_2\rrbracket
\end{array}
$$

$$
\begin{array}{lcl}
\Psi\llbracket\texttt{var}\ v\rrbracket & = & v \\
\Psi\llbracket\underline{\underline{\lambda}}_i F\rrbracket & = & \underline{\lambda}_i v^\diamond.\Psi\llbracket F v^\diamond\rrbracket \\
\Psi\llbracket E_1\underline{\underline{@}}_i E_2\rrbracket & = & \Psi\llbracket E_1\rrbracket\underline{@}_1 \Psi\llbracket E_2\rrbracket \\
\Psi\llbracket\underline{\underline{\texttt{let}}}_i\ E\ \underline{\underline{\texttt{in}}}\ F\rrbracket & = & \underline{\texttt{let}}_i\ v^\diamond = \Psi\llbracket E\rrbracket\ \underline{\texttt{in}}\ \Psi\llbracket F v^\diamond\rrbracket
\end{array}
$$

We extend the above mappings Φ and Ψ compatible to the metalanguage, *i.e.* application of Φ to $e \in Expr$ changes the syntax constructors which appear in e from $MExpr$ to $MExpr'$. By induction on multi-level terms we can prove the obvious connection between first-order and higher-order abstract syntax:

Lemma 1 *1.* $\Phi(\Psi E) = E$ *for all* $E \in MExpr'$ *and*

2. $\Psi(\Phi E) = E$ *for all* $E \in MExpr$.

Transforming our example to HOAS yields

$$\underline{\underline{\lambda}}_1 \lambda x.(\underline{\underline{\texttt{let}}}_1\ a\underline{\underline{@}}_1 b\ \underline{\underline{\texttt{in}}}\ \lambda v.(\underline{\underline{\lambda}}_0 \lambda z.z))\underline{\underline{@}}_0 x.$$

Writing the reducer $\mathcal{R}'_c\llbracket\ \rrbracket$ for higher-order abstract syntax and with the continuation abstraction transferred to the right sides we need to make explicit the variable constructor $\texttt{var}\ : MExpr' \to MExpr'$ which serves to stop the reducer from processing variable bindings. The resulting reducer is shown in Fig. 3. Our example is transformed as follows:

$$
\begin{array}{c}
\mathcal{R}'_c\llbracket\underline{\underline{\lambda}}_1 \lambda x.(\underline{\underline{\texttt{let}}}_1\ a\underline{\underline{@}}_1 b\ \underline{\underline{\texttt{in}}}\ \lambda v.(\underline{\underline{\lambda}}_0 \lambda z.z))\underline{\underline{@}}_0 x\rrbracket(\lambda z.z) \\
= \underline{\underline{\lambda}}_0 \lambda x.\underline{\underline{\texttt{let}}}_0\ a\underline{\underline{@}}b\ \underline{\underline{\texttt{in}}}\ \lambda v.x.
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{R}_c : \mathit{MExpr} \rightarrow \mathit{PEEnv} \rightarrow \mathit{PECont} \rightarrow \mathit{PEVal} & & \\
\mathcal{R}_c\llbracket v \rrbracket \rho\kappa & = & \kappa(\rho\llbracket v \rrbracket) \\
\mathcal{R}_c\llbracket \underline{\lambda}_0 v.E \rrbracket \rho\kappa & = & \kappa(\lambda y.\mathcal{R}_c\llbracket E \rrbracket \rho[v \mapsto y]) \\
\mathcal{R}_c\llbracket E_1 \underline{@}_0 E_2 \rrbracket \rho\kappa & = & \mathcal{R}_c\llbracket E_1 \rrbracket \rho(\lambda y_1.\mathcal{R}_c\llbracket E_2 \rrbracket \rho(\lambda y_2.(y_1\ y_2)\ \kappa)) \\
\mathcal{R}_c\llbracket \underline{\mathtt{let}}_0\ v = E_1\ \underline{\mathtt{in}}\ E_2 \rrbracket \rho\kappa & = & \mathcal{R}_c\llbracket E_1 \rrbracket \rho(\lambda y.\mathcal{R}_c\llbracket E_2 \rrbracket \rho[v \mapsto y]\kappa) \\
\mathcal{R}_c\llbracket \underline{\lambda}_{i+1} v.E \rrbracket \rho\kappa & = & \kappa(\underline{\lambda}_i v^\diamond.\mathcal{R}_c\llbracket E \rrbracket \rho[v \mapsto v^\diamond](\lambda z.z)) \\
\mathcal{R}_c\llbracket E_1 \underline{@}_{i+1} E_2 \rrbracket \rho\kappa & = & \mathcal{R}_c\llbracket E_1 \rrbracket \rho(\lambda E_1'.\mathcal{R}_c\llbracket E_2 \rrbracket \rho(\lambda E_2'.\kappa(E_1' \underline{@}_i E_2'))) \\
\mathcal{R}_c\llbracket \underline{\mathtt{let}}_{i+1}\ v = E_1\ \underline{\mathtt{in}}\ E_2 \rrbracket \rho\kappa & = & \mathcal{R}_c\llbracket E_1 \rrbracket \rho(\lambda E_1'.\underline{\mathtt{let}}_i\ v^\diamond = E_1'\ \underline{\mathtt{in}}\ \mathcal{R}_c\llbracket E_2 \rrbracket \rho[v \mapsto v^\diamond]\kappa)
\end{array}
$$

Figure 2: Multi-level Continuation-Based Reducer

$$
\begin{array}{lcl}
\mathcal{R}_c' : \mathit{MExpr}' \rightarrow \mathit{PECont} \rightarrow \mathit{PEVal} & & \\
\mathcal{R}_c'\llbracket \mathtt{var}\ E \rrbracket & = & \lambda\kappa.\kappa E \\
\mathcal{R}_c'\llbracket \underline{\underline{\lambda}}_0 F \rrbracket & = & \lambda\kappa.\kappa(\lambda y.\mathcal{R}_c'\llbracket Fy \rrbracket) \\
\mathcal{R}_c'\llbracket E_1 \underline{\underline{@}}_0 E_2 \rrbracket & = & \lambda\kappa.\mathcal{R}_c'\llbracket E_1 \rrbracket(\lambda y_1.\mathcal{R}_c'\llbracket E_2 \rrbracket(\lambda y_2.(y_1\ y_2)\ \kappa)) \\
\mathcal{R}_c'\llbracket \underline{\underline{\mathtt{let}}}_0\ E_1\ \underline{\underline{\mathtt{in}}}\ F \rrbracket & = & \lambda\kappa.\mathcal{R}_c'\llbracket E_1 \rrbracket(\lambda y.\mathcal{R}_c'\llbracket Fy \rrbracket\kappa) \\
\mathcal{R}_c'\llbracket \underline{\underline{\lambda}}_{i+1} F \rrbracket & = & \lambda\kappa.\kappa(\underline{\underline{\lambda}}_i \lceil \lambda v^\diamond.\langle \mathcal{R}_c'\llbracket Fv^\diamond \rrbracket(\lambda z.z)\rangle \rceil) \\
\mathcal{R}_c'\llbracket E_1 \underline{\underline{@}}_{i+1} E_2 \rrbracket & = & \lambda\kappa.\mathcal{R}_c'\llbracket E_1 \rrbracket(\lambda E_1'.\mathcal{R}_c'\llbracket E_2 \rrbracket(\lambda E_2'.\kappa(E_1' \underline{\underline{@}}_i E_2'))) \\
\mathcal{R}_c'\llbracket \underline{\underline{\mathtt{let}}}_{i+1}\ E_1\ \underline{\underline{\mathtt{in}}}\ F \rrbracket & = & \lambda\kappa.\mathcal{R}_c'\llbracket E_1 \rrbracket(\lambda E_1'.\underline{\underline{\mathtt{let}}}_i\ E_1'\ \underline{\underline{\mathtt{in}}}\ \lceil \lambda v^\diamond.\langle \mathcal{R}_c'\llbracket Fv^\diamond \rrbracket\kappa\rangle \rceil)
\end{array}
$$

Figure 3: Multi-level Continuation-Based Reducer for HOAS

The following lemma captures the correspondence between the reducers $\mathcal{R}_c'\llbracket\_\rrbracket$ and $\mathcal{R}_c\llbracket\_\rrbracket$. Its proof is found in Appendix A.

Lemma 2 *Let $E \in \mathit{MExpr}$, $\kappa \in \mathit{PECont}$ well-behaved [5], and $\rho = [v_i \mapsto E_i \mid 1 \leq i \leq m]$ such that $FV(E) \subseteq \{v_1, \ldots, v_m\}$. Let further κ^Φ be defined by $\kappa^\Phi(E') = \Phi(\kappa(\Psi E'))$ for all $E' \in \mathit{MExpr}'$. Then*

$$\Phi(\mathcal{R}_c\llbracket E \rrbracket \rho\kappa) = \mathcal{R}_c'\llbracket \Phi(E)[v_i := \Phi(E_i)] \rrbracket \kappa^\Phi.$$

Roughly, a specialization-time continuation is well-behaved [5] if it coincides with an evaluation-time continuation and does not capture any variable bindings of the source program. All continuations introduced by our continuation-based reducers are well-behaved.

5.2 A Cogen for Multi-Level Continuation-Based Reduction

Only one thing in the definition still keeps us from replacing $\mathcal{R}_c'\llbracket\_\rrbracket$ with standard evaluation and considering the equations as combinator definitions: A variable v is not simply interpreted as an environment lookup but as a continuation returning the value: $\lambda\kappa.\kappa v$. However, there's a simple and correct fix: Bind the variable to the continuation from the beginning. In the transition to standard evaluation we have to replace every occurrence of FE by $F(\lambda\kappa.\kappa E)$. And that is all!

To obtain a generating extension we make the HOAS constructors executable. The types of their executable counterparts and their defining equations are specified in Fig. 4, using $\eta = \lambda x.\lambda\kappa.\kappa x$ and $\iota = \lambda z.z$. The relationship to $\mathcal{R}_c'\llbracket\_\rrbracket$ is stated as follows.

Lemma 3 *For all $E \in \mathit{MExpr}'$, $\kappa \in \mathit{PECont}$:*

$$\mathcal{R}_c'\llbracket E[v_i := E_i] \rrbracket \kappa = (E[v_i := \eta E_i])\kappa.$$

The proof is found in Appendix B. Taking Lemma 2 and Lemma 3 together we obtain the correctness of the PGG just constructed.

Theorem 1 *Let $E \in \mathit{MExpr}$, $\kappa \in \mathit{PECont}$ well-behaved, and $\rho = [v_i \mapsto E_i \mid 1 \leq i \leq m]$ such that $FV(E) \subseteq \{v_1, \ldots, v_m\}$. Let further κ^Φ be defined by $\kappa^\Phi(E') = \Phi(\kappa(\Psi(E')))$ for all $E' \in \mathit{MExpr}'$. Then*

$$\Phi(\mathcal{R}_c\llbracket E \rrbracket \rho\kappa) = \Phi(E)[v_i := \eta(\Phi(E_i))]\kappa^\Phi.$$

To complete our running example, we can now write down the program generator which is the generating extension for our example.

$$\underline{\underline{\lambda}}_1 \lambda x.(\underline{\underline{\mathtt{let}}}_1\ a \underline{@}_1 b\ \underline{\underline{\mathtt{in}}}\ \lambda v.(\underline{\underline{\lambda}}_0 \lambda z.z))\underline{\underline{@}}_0 x$$

It is actually just the multi-level program in HOAS. Running the program with the interpretation in Fig. 4 yields:

$$\underline{\underline{\lambda}}_0 \lambda x.\underline{\underline{\mathtt{let}}}_0\ a \underline{\underline{@}} b\ \underline{\underline{\mathtt{in}}}\ \lambda v.x$$

6 Using Control Operators

Lawall and Danvy [24] show how to implement continuation-based reduction in direct style (DS) using control operators. In an extension of that work, Lawall and Danvy [25] present the first hand-written PGG performing continuation-based reduction in direct style. Both exploit the control operators **shift** and **reset** [12]. **shift** and **reset** provide composable continuations: **reset** delimits a continuation and **shift** abstracts the continuation up to the innermost lexically enclosing **reset**. A continuation abstracted with **shift** is composed with the current continuation, in contrast to continuations provided by **call/cc** which discard the current continuation.

$$
\begin{array}{lcl}
\underline{\underline{\underline{\lambda}}} & : & \mathit{Level} \times (\mathit{PECont} \to \mathit{PECont}) \to \mathit{PECont} \to \mathit{PEVal} \\
\underline{\underline{\underline{@}}} & : & \mathit{Level} \times \mathit{PECont} \times \mathit{PECont} \to \mathit{PECont} \to \mathit{PEVal} \\
\underline{\underline{\underline{\mathtt{let}}}} & : & \mathit{Level} \times \mathit{PECont} \times (\mathit{PECont} \to \mathit{PECont}) \to \mathit{PECont} \to \mathit{PEVal}
\end{array}
$$

$$
\begin{array}{lcl}
\underline{\underline{\underline{\lambda}}}_0(f) & = & \lambda\kappa.\kappa\lambda y.f(\eta y) \\
\underline{\underline{\underline{@}}}_0(E_1, E_2) & = & \lambda\kappa.E_1(\lambda e_1.E_2(\lambda e_2.(e_1 e_2)\kappa)) \\
\underline{\underline{\underline{\mathtt{let}}}}_0(E_1, f) & = & \lambda\kappa.E_1(\lambda e.f(\eta e)\kappa) \\
\underline{\underline{\underline{\lambda}}}_{i+1}(f) & = & \lambda\kappa.\kappa\lceil\underline{\underline{\underline{\lambda}}}_{\langle i\rangle}(\lambda v^\diamond.\langle f(\eta v^\diamond)\iota\rangle)\rceil \\
\underline{\underline{\underline{@}}}_{i+1}(E_1, E_2) & = & \lambda\kappa.E_1(\lambda E_1'.E_2(\lambda E_2'.\kappa\lceil\underline{\underline{\underline{@}}}_{\langle i\rangle}(\langle E_1'\rangle, \langle E_2'\rangle)\rceil)) \\
\underline{\underline{\underline{\mathtt{let}}}}_{i+1}(E_1, f) & = & \lambda\kappa.E_1(\lambda E_1'.\lceil\underline{\underline{\underline{\mathtt{let}}}}_{\langle i\rangle}(\langle E_1'\rangle, \lambda v^\diamond.\langle f(\eta v^\diamond)\kappa\rangle)\rceil)
\end{array}
$$

Figure 4: Multi-level Continuation-Based PGG

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathcal{R}_d : \mathit{MExpr} \to \mathit{PEEnv} \to \mathit{PEVal}} \\
\mathcal{R}_d[\![v]\!]\rho & = & \rho[\![v]\!] \\
\mathcal{R}_d[\![\underline{\lambda}_0 v.E]\!]\rho & = & \lambda y.\mathcal{R}_d[\![E]\!]\rho[v \mapsto y] \\
\mathcal{R}_d[\![E_1 \underline{@}_0 E_2]\!]\rho & = & \mathcal{R}_d[\![E_1]\!]\rho(\mathcal{R}_d[\![E_2]\!]\rho) \\
\mathcal{R}_d[\![\underline{\mathtt{let}}\ v = E_1\ \underline{\mathtt{in}}\ E_2]\!]\rho & = & \mathcal{R}_d[\![E_2]\!]\rho[v \mapsto (\mathcal{R}_d[\![E_1]\!]\rho)] \\
\mathcal{R}_d[\![\underline{\lambda}_{i+1} v.E]\!]\rho & = & \underline{\lambda}_i v^\diamond.\mathbf{reset}(\mathcal{R}_d[\![E]\!]\rho[v \mapsto v^\diamond]) \\
\mathcal{R}_d[\![E_1 \underline{@}_{i+1} E_2]\!]\rho & = & (\mathcal{R}_d[\![E_1]\!]\rho \underline{@}_i \mathcal{R}_d[\![E_2]\!]\rho) \\
\mathcal{R}_d[\![\underline{\mathtt{let}}_{i+1}\ v = E_1\ \underline{\mathtt{in}}\ E_2]\!]\rho & = & \mathbf{shift}\ k.\underline{\mathtt{let}}_i\ v^\diamond = \mathcal{R}_d[\![E_1]\!]\rho\ \underline{\mathtt{in}}\ \mathbf{reset}(k(\mathcal{R}_d[\![E_2]\!]\rho[v \mapsto v^\diamond]))
\end{array}
$$

Figure 5: Multi-level Continuation-Based Reducer in Direct Style

We provide the standard definition in terms of a CPS transformation $\mathcal{T}$ in Appendix C. Using them we can write the multi-level reducer $\mathcal{R}_d$ in DS as shown in Fig. 5. $\mathcal{R}_c$ and $\mathcal{R}_d$ are related by the CPS transformation.

Lemma 4 $\mathcal{T}[\![\mathcal{R}_d[\![\_]\!]]\!] = \mathcal{R}_c[\![\_]\!]$.

If we again make the transition to HOAS, as demonstrated for the CPS version, the result has an almost immediate interpretation as a PGG. The final result is shown in Fig. 6. Again, the two variant interpretations of the HOAS constructors are related via the CPS transformation. For the DS generator we have the following convenient result which is proved by CPS-transforming it to Theorem 1.

Lemma 5 *For all $E \in \mathit{MExpr}$:*

$$\mathcal{R}_d[\![E]\!][v_i \mapsto E_i] = \Phi(E)[v_i := \Phi(E_i)].$$

6.1 Higher-Level shift and reset?

As **shift** and **reset** can be generalized to abstract more than one context, it is natural to ask whether multi-level **shift** and **reset**s may be used to advantage in multi-level generating extensions.

In a program generated by continuation-based specialization no subexpressions of the forms $(\mathtt{let}\ v = e\ \mathtt{in}\ e)@e$ and $e@(\mathtt{let}\ v = e\ \mathtt{in}\ e)$ arise. All `let` expressions aggregate inside of lambdas and on the outermost level, *i.e.* residual expressions have the form N where

$$
\begin{array}{lcl}
N & ::= & N' \mid \underline{\mathtt{let}}_i\ v = N\ \underline{\mathtt{in}}\ N \\
N' & ::= & v \mid N' \underline{@}_i N' \mid \underline{\lambda}_i v.N
\end{array}
$$

However, the nesting order of dynamic `let` expressions is not changed with respect to the subject program. If we would simply use $\mathbf{shift}_i$ and $\mathbf{reset}_i$ in the equation for $\underline{\underline{\underline{\mathtt{let}}}}_i$ and the highest level **reset** in all equations for $\underline{\underline{\underline{\lambda}}}$ we would effectively exchange `let` binders. For example, $\mathbf{shift}_2$ in the equation for $\underline{\underline{\underline{\mathtt{let}}}}_i$ (for $i \geq 2$) would move $\underline{\underline{\underline{\mathtt{let}}}}_2$ binders outside of $\underline{\underline{\underline{\mathtt{let}}}}_1$ binders. As such a transformation can break variable bindings we conclude that $\mathbf{shift}_2$ is not applicable in general.

6.2 Why the PGG need not tamper with control

In other approaches dealing with PGGs for continuation-based specialization [8, 25], the PGG itself needs to manipulate control. This way, more static reductions can be performed at generation time. However, the additional complexity is considerable. In our approach, multi-level specialization does the job: The PGG generates the program *with every level annotation incremented by one.* A first generation run without any static input performs the generation-time reductions and yields the program generator ready to receive the static inputs of the first level.

7 Implementation

The development presented in the preceding sections extends smoothly to the full effect-free subset of Scheme. We illustrate this by giving some samples from our library. In all the example functions, the parameter `lv` denotes the level argument.

7.1 PGG library functions in CPS

Application The library code for function application accepts a function continuation `fc` and a list of argument continuations `A*c`.

$$
\begin{array}{lcl}
\underline{\underline{\lambda}} & : & \mathit{Level} \times (\mathit{MExpr} \to \mathit{MExpr}) \to \mathit{MExpr} \\
\underline{\underline{@}} & : & \mathit{Level} \times \mathit{MExpr} \times \mathit{MExpr} \to \mathit{MExpr} \\
\underline{\underline{\mathtt{let}}} & : & \mathit{Level} \times \mathit{MExpr} \times (\mathit{MExpr} \to \mathit{MExpr}) \to \mathit{MExpr}
\end{array}
$$

$$
\begin{array}{lcl}
\underline{\underline{\lambda}}_0 f & = & f \\
\underline{\underline{@}}_0 (e_1, e_2) & = & e_1 \; e_2 \\
\underline{\underline{\mathtt{let}}}_0 (e, f) & = & f \; e \\
\underline{\underline{\lambda}}_{i+1} f & = & \lceil \underline{\underline{\lambda}}_{\langle i \rangle} \lambda v^{\diamond}.\langle \mathbf{reset}(f v^{\diamond})\rangle \rceil \\
\underline{\underline{@}}_{i+1} (e_1, e_2) & = & \lceil \underline{\underline{@}}_{\langle i \rangle} (\langle e_1 \rangle, \langle e_2 \rangle) \rceil \\
\underline{\underline{\mathtt{let}}}_{i+1} (e, f) & = & \mathbf{shift}\; k.\lceil \underline{\underline{\mathtt{let}}}_{\langle i \rangle} (\langle e \rangle, \lambda v^{\diamond}.\langle \mathbf{reset}(k(f v^{\diamond}))\rangle) \rceil
\end{array}
$$

Figure 6: Multi-level Continuation-Based PGG in Direct Style

```
(define (_app lv fc . A*c)
  (lambda (kappa)
    (fc
     (lambda (f)
       (process-arg-list
        A*c
        (lambda (A*)
          (if (zero? lv)
              ((apply f A*) kappa)
              (kappa
               `(_APP ,(- lv 1) ,f ,@A*))))))))))
```

It makes use of another library function `process-arg-list` which processes the list of argument continuations.

Lambda Abstraction Lambda abstraction accepts the arity `arity` of the lambda (a list of symbols in order to generate readable residual programs) and a function `f` which accepts the list of abstracted variables and produces a continuation which constructs the body of the lambda.

```
(define (_lambda lv arity f)
  (lambda (kappa)
    (let* ((V*  (map gensym arity))
           (V*c (map result V*))
           (fun `(LAMBDA ,V*
                  ,((apply f V*c) id))))
      (if (= lv 1)
          (kappa fun)
          (kappa
           `(_LAMBDA ,(- lv 1) ',arity ,fun))))))
```

The function `result` is $\eta = \lambda x.\lambda\kappa.\kappa x$ and `id` is the identity function.

Let Expression A `let` expression accepts the `let` header expression continuation `e` and a function `f` with the same meaning as for lambda abstractions.

```
(define (_let lv e f)
  (lambda (kappa)
    (let* ((V  (gensym 'fresh))
           (Vc (result V)))
      (e
       (lambda (value)
         (cond
          ((zero? lv)
           ((f (result value)) kappa))
          (else
           `(_LET ,(- lv 1) ,value
                  (LAMBDA (,V)
                    ,((f Vc) kappa))))))))))
```

Primitive Operators The interpretation of primitive operators accepts the operator `op` (a symbol, if `lv` > 0; otherwise the operator itself) and a list `A*c` of argument continuations.

```
(define (_op lv op . A*c)
  (lambda (kappa)
    (process-arg-list
     A*c
     (lambda (A*)
       (kappa
        (cond
         ((zero? lv)
          (apply op A*))
         ((= 1 lv)
          `(_OP 0 ,op ,@A*))
         (else
          `(_OP ,(- lv 1) ',op ,@A*))))))))
```

Conditional The conditional accepts three continuations.

```
(define (_if lv e1c e2c e3c)
  (lambda (kappa)
    (e1c (lambda (e1)
           (if (zero? lv)
               (if e1 (e2c kappa) (e3c kappa))
               `(_IF ,(- lv 1) ,e1
                                ,(e2c kappa)
                                ,(e3c kappa)))))))
```

7.2 PGG library functions in DS

As expected, the library functions presented below get even simpler in DS. However, there is a slight complication. In the implementation of the conditional we require that the specializer controls the continuation. The solution is discussed in the respective paragraph below.

Application This function is identical to the implementation of Glück and Jørgensen [16]. It is only included for completeness.

```
(define (_app lv f . args)
  (if (= lv 1)
      `(,f ,@args)
      `(_APP ,(- lv 1) ,f ,@args)))
```

Lambda Abstraction Here, we are fortunate because the higher-order nature of the abstract syntax gives us control over the continuation of the specialization of the lambda's body. As the specialization is wrapped in the function `f` (which accepts a list of variables and returns the specialized body) we can easily wrap a **reset** around its execution.

```
(define (_lambda lv arity f)
  (let* ((vars (map gensym arity))
         (fun `(LAMBDA ,vars
                 ,(reset (apply f vars)))))
    (if (= lv 1)
        fun
        `(_LAMBDA ,(- lv 1) ',arity ,fun))))
```

Let Expression The same situation arises as in the previous case. Due to HOAS we are in control of the body's specialization continuation.

```
(define (_let lv e f)
  (let ((var (gensym 'fresh)))
    (cond
     ((zero? lv)
      (f e))
     (else
      (shift k
        `(_LET ,(- lv 1)
               ,e
               (LAMBDA (,var)
                 ,(reset (k (f var))))))))))
```

Primitive Operators Taken literally from [16].

```
(define (_op lv op . args)
  (if (= lv 1)
      `(,op ,@args)
      `(_OP ,(- lv 1) ',op ,@args)))
```

Conditional At the conditional, the specializer must be able to control the continuations of the then and else branches. This is trivial to achieve in the CPS version. In DS it requires to change the abstract syntax so that the branches of the conditional are *thunks* returning their specialized code. This way, we retain control over their specialization continuation and can insert the appropriate **reset**s.

```
(define (_if lv e1 et2 et3)
  (if (= lv 1)
      `(IF ,e1 ,(reset (et2)) ,(reset (et3)))
      `(_IF ,(- lv 1)
            ,e1
            (LAMBDA () ,(reset (et2)))
            (LAMBDA () ,(reset (et3))))))
```

8 Results

Here, we report some figures of our implementation. It handles the functional subset of Scheme augmented with user-defined (partially static) data structures. It consists of a preprocessing phase, a constraint-based binding-time analysis [9], the multi-level code generation, and the run-time libraries in CPS and in DS. The run-time libraries are constructed as outlined in Sections 5 and 6. The DS version uses Filinski's implementation of **shift** and **reset** [14]. Figure 7 shows a size summary of the program.

In order to substantiate the claim that hand-written PGGs are more efficient we have performed a comparison with Similix [6]. We have used three subject programs "app" (standard list append), "ctors" (testing partially static data), and "lambda" (partially static functions). All times are mean values over 100 runs on an IBM 320 with 32MB main memory using Scheme48 [21]. Timings are also shown in Fig. 7. The construction of the generation extension with our PGG is almost an order of magnitude faster. The PGG constructed generating extensions in direct style are faster by a factor of two to five, the CPS versions by a factor of one to six.

Furthermore, we have performed another comparison using realistic applications. We have applied Similix and our DS PGG to generate LR parsers (lrpars) [31] and online compiler generators (ocogen) [30]. The results are presented in Fig. 8.

The time differences with respect to Similix are partly due to two facts: There are additional static analysis phases (which add to the preprocessing time) and there is a sophisticated postprocessing pass (which adds to the specialization time).

9 Related Work

Context propagation for two-level programs can be achieved in several ways. The basic idea appears in Danvy and Filinski's work on a one-pass CPS transformation [12]. Consel and Danvy [10] suggest to transform the subject program into CPS, Bondorf [5] writes the specializer in non-standard CPS—in essentially the same style as [12]—, and Lawall and Danvy [24] use control operators in the specializer.

Hand-written PGGs are reported in several papers for a variety of languages [23, 2, 3]. The context propagation ideas can be applied to hand-written PGGs: Bondorf and Dussart [8] construct and prove correct a hand-written continuation-based PGG in CPS, Lawall and Danvy [25] present several continuation-based PGGs in direct style. None of them considers multi-level specialization.

Glück and Jørgensen [16] consider PGGs for multi-level specialization, but they do not cover continuation-based specialization. Their implementation is also library-based but the library functions are constructed in an ad-hoc manner.

Higher-order abstract syntax has been reported by Pfenning and Elliot [28] as a tool to alleviate program transformations from explicitly dealing with bindings, scoping, and substitution issues. Mogensen has used it to write self-interpreters and self-applicable online partial evaluators for the lambda calculus [26, 27]. Our work is another indication of the power of higher-order abstract syntax in program transformations.

10 Conclusion

We have presented the first PGG for multi-level continuation-based specialization. We have shown that continuation-based specialization extends smoothly to

program part	lines	bytes
auxiliary definitions	462	13543
preprocessing	525	15713
binding-time analysis	500	16128
code generation	114	3911
common library	129	4178
DS interpretation	140	3847
CPS interpretation	294	8428
total	2164	65748

	program	app	ctors	lambda
Similix	binding-time analysis	545	1156	1195
	PGG construction	1601	4870	6113
	specialization	57	202	182
our PGG	binding-time analysis	160	470	427
	PGG construction (DS)	48	97	121
	specialization (DS)	30	37	109
	PGG construction (CPS)	62	127	147
	specialization (CPS)	61	93	34

Figure 7: Sizes and comparative run times (in ms) of our PGG

program	size	preprocess	construct generating extension	specialize
lrpars	793	15.95	87.69	84.55
	PGG:	11.81	1.71	35.42
ocogen	2311	57.57	304.01	132.85
	PGG:	33.72	3.94	34.86

all sizes in cons cells, times in seconds (first line Similix, second line PGG)

Figure 8: Realistic Examples

multi-level specialization. Using our calculational approach it is easy to derive a PGG from a specializer. Compared to previous presentations, ours is much simpler and yields the same results.

We believe that it is simpler to write PGGs by hand than writing self-applicable specializers, as we do not have to be wary of binding times while we construct the PGG and as we can freely exploit all features of the language both in the PGG and in the generated generating extensions. It is also beneficial from an efficiency standpoint.

Acknowledgements

Thanks are due to Michael Sperber for reading drafts of this paper and to the referees for their valuable comments.

References

[1] L. O. Andersen. Self-applicable C program specialization. In C. Consel, editor, *Workshop Partial Evaluation and Semantics-Based Program Manipulation '92*, pages 54–61, San Francisco, CA, June 1992. Yale University. Report YALEU/DCS/RR-909.

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, Dept. of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 København Ø, May 1994.

[3] L. Birkedal and M. Welinder. Partial evaluation of Standard ML. Rapport 93/22, DIKU, University of Copenhagen, 1993.

[4] A. Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Programming*, 17:3–34, 1991.

[5] A. Bondorf. Improving binding-times without explicit CPS conversion. In *Proc. Conference on Lisp and Functional Programming*, pages 1–10, San Francisco, CA, USA, June 1992.

[6] A. Bondorf. *Similix 5.0 Manual.* DIKU, University of Copenhagen, May 1993.

[7] A. Bondorf and O. Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Programming*, 19(2):151–195, 1991.

[8] A. Bondorf and D. Dussart. Improving CPS-based partial evaluation: Writing cogen by hand. In P. Sestoft and H. Søndergaard, editors, *Workshop Partial Evaluation and Semantics-Based Program Manipulation '94*, pages 1–10, Orlando, Fla., June 1994. ACM.

[9] A. Bondorf and J. Jørgensen. Efficient analysis for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, July 1993.

[10] C. Consel and O. Danvy. For a better support of static data flow. In Hughes [18], pages 496–519. LNCS 523.

[11] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proc. 20th ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, Jan. 1993. ACM Press.

[12] O. Danvy and A. Filinski. Abstracting control. In *Proc. Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, 1990. ACM.

[13] A. de Niel. *Self-Applicable Partial Evaluation of Polymorphically Typed Functional Languages.* PhD thesis, Katholieke Universiteit Leuven, Leuven, Belgium, Jan. 1993.

[14] A. Filinski. Representing monads. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pages 446–457, Portland, OG, Jan. 1994. ACM Press.

[15] Y. Futamura. Partial evaluation of computation process — an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[16] R. Glück and J. Jørgensen. Efficient multi-level generating extensions for program specialization. In D. Swierstra and M. Hermenegildo, editors, *Programming Languages, Implementations, Logics, and Programs (PLILP '95)*, Utrecht, The Netherlands, Sept. 1995. Springer-Verlag.

[17] C. K. Holst. Syntactic currying. student report, DIKU, 1989.

[18] J. Hughes, editor. *Functional Programming Languages and Computer Architecture*, Cambridge, MA, 1991. Springer-Verlag. LNCS 523.

[19] Institute of Electrical and Electronic Engineers, Inc. IEEE standard for the Scheme programming language. IEEE Std 1178-1990, New York, NY, 1991.

[20] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[21] R. A. Kelsey and J. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1994.

[22] J. Launchbury. A strongly-typed self-applicable partial evaluator. In Hughes [18], pages 145–164. LNCS 523.

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

[24] J. Lawall and O. Danvy. Continuation-based partial evaluation. In *Proc. Conference on Lisp and Functional Programming*, pages 227–238, Orlando, Fla, USA, June 1994. ACM Press.

[25] J. Lawall and O. Danvy. Continuation-based partial evaluation. Extended version of [24] from ftp://ftp.daimi.aau.dk/pub/danvy/Papers/, Jan. 1995.

[26] T. Æ. Mogensen. Efficient self-interpretation in lambda calculus. *Journal of Functional Programming*, 2(3):345–364, July 1992.

[27] T. Æ. Mogensen. Self-applicable online partial evaluation of pure lambda calculus. In Scherlis [29], pages 39–44.

[28] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proc. Conference on Programming Language Design and Implementation '88*, pages 199–208, Atlanta, July 1988. ACM.

[29] W. Scherlis, editor. *ACM SIGPLAN Symp. Partial Evaluation and Semantics-Based Program Manipulation '95*, La Jolla, CA, June 1995. ACM Press.

[30] M. Sperber. Self-applicable online partial evaluation. Submitted for publication, Dec. 1995.

[31] M. Sperber and P. Thiemann. The essence of LR parsing. In Scherlis [29], pages 146–155.

A Proof of Lemma 2

We prove by induction on E that

$$\Phi(\mathcal{R}_c\llbracket E\rrbracket\rho\kappa) = \mathcal{R}'_c\llbracket\Phi(E)[v_i := \Phi(E_i)]\rrbracket\kappa^\Phi)$$

where $\kappa^\Phi(E') = \Phi(\kappa(\Psi(E')))$ for all $E' \in \mathit{MExpr}'$.

case v_j

$$\begin{aligned}
& \Phi(\mathcal{R}_c\llbracket v_j\rrbracket\rho\kappa) \\
= \quad & \text{definition of } \mathcal{R}_c \\
& \Phi(\kappa(\rho\llbracket v_j\rrbracket)) \\
= \quad & \text{assumption on } \rho \\
& \Phi(\kappa E_j) \\
= \quad & \text{definition of } \kappa^\Phi \\
& \kappa^\Phi(\Phi(E_j)) \\
= \quad & \text{definition of } \mathcal{R}'_c \\
& \mathcal{R}'_c\llbracket \mathtt{var}\ (\Phi E_j)\rrbracket\kappa^\Phi \\
= \quad & \text{substitution} \\
& \mathcal{R}'_c\llbracket(\mathtt{var}\ v_j)[v_i := \Phi E_j]\rrbracket\kappa^\Phi \\
= \quad & \text{definition of } \mathcal{R}'_c \\
& \mathcal{R}'_c\llbracket(\Phi v_j)[v_i := \Phi E_j]\rrbracket\kappa^\Phi
\end{aligned}$$

case $E_1\underline{@}_0E_2$

$$\begin{aligned}
& \mathcal{R}'_c\llbracket\Phi(E_1\underline{@}_0E_2)[\ldots]\rrbracket\kappa^\Phi \\
= \quad & \text{definition of } \Phi \\
& \mathcal{R}'_c\llbracket(\Phi(E_1)\underline{\underline{@}}_0\Phi(E_2))[\ldots]\rrbracket\kappa^\Phi \\
= \quad & \text{substitution} \\
& \mathcal{R}'_c\llbracket\Phi(E_1)[\ldots]\underline{\underline{@}}_0\Phi(E_2)[\ldots]\rrbracket\kappa^\Phi \\
= \quad & \text{definition of } \mathcal{R}'_c \\
& \mathcal{R}'_c\llbracket\Phi(E_1)[\ldots]\rrbracket(\lambda y_1.\mathcal{R}'_c\llbracket\Phi(E_2)[\ldots]\rrbracket(\lambda y_2.y_1y_2\kappa^\Phi)) \\
= \quad & \text{induction hypothesis} \\
& \Phi(\mathcal{R}_c\llbracket E_1\rrbracket\rho(\lambda y_1.\Psi(\mathcal{R}'_c\llbracket\Phi(E_2)[\ldots]\rrbracket(\lambda y_2.\Phi(y_1)y_2\kappa^\Phi)))) \\
= \quad & \text{induction hypothesis} \\
& \Phi(\mathcal{R}_c\llbracket E_1\rrbracket\rho(\lambda y_1.\Psi(\Phi\mathcal{R}_c\llbracket(E_2)\rrbracket\rho(\lambda y_2.\Psi(\Phi(y_1)\Phi(y_2)\kappa^\Phi))))) \\
= \quad & \text{cancellation } \Psi\Phi \\
& \Phi(\mathcal{R}_c\llbracket E_1\rrbracket\rho(\lambda y_1.\mathcal{R}_c\llbracket(E_2)\rrbracket\rho(\lambda y_2.\Psi(\Phi(y_1)\Phi(y_2)\kappa^\Phi)))) \\
= \quad & \text{definition of } \kappa^\Phi \\
& \Phi(\mathcal{R}_c\llbracket E_1\rrbracket\rho(\lambda y_1.\mathcal{R}_c\llbracket(E_2)\rrbracket\rho(\lambda y_2.\Psi(\Phi(y_1y_2\kappa))))) \\
= \quad & \text{cancellation } \Psi\Phi \\
& \Phi(\mathcal{R}_c\llbracket E_1\rrbracket\rho(\lambda y_1.\mathcal{R}_c\llbracket(E_2)\rrbracket\rho(\lambda y_2.y_1y_2\kappa))) \\
= \quad & \text{definition of } \mathcal{R}_c \\
& \Phi(\mathcal{R}_c\llbracket E_1\underline{@}_0E_2\rrbracket\rho\kappa)
\end{aligned}$$

case $\underline{\mathtt{let}}_{i+1}\ v = E_1\ \underline{\mathtt{in}}\ E_2$

$$\begin{aligned}
& \mathcal{R}'_c\llbracket\Phi(\underline{\mathtt{let}}_{i+1}\ v = E_1\ \underline{\mathtt{in}}\ E_2)[\ldots]\rrbracket\kappa^\Phi \\
= \quad & \text{substitution and definition of } \Phi \\
& \mathcal{R}'_c\llbracket\underline{\underline{\mathtt{let}}}_{i+1}\ \Phi(E_1)[\ldots]\ \underline{\underline{\mathtt{in}}}\ \lambda v.\Phi(E_2)[\ldots]\rrbracket\kappa^\Phi \\
= \quad & \text{definition of } \mathcal{R}'_c \\
& \mathcal{R}'_c\llbracket\Phi(E_1)[\ldots]\rrbracket(\lambda E'_1.\underline{\underline{\mathtt{let}}}_i\ E'_1\ \underline{\underline{\mathtt{in}}} \\
& \quad \lambda v^\diamond.\mathcal{R}'_c\llbracket(\lambda v.\Phi(E_2)[\ldots])v^\diamond\rrbracket\kappa^\Phi) \\
= \quad & \text{induction, } \beta \\
& \Phi(\mathcal{R}_c\llbracket E_1\rrbracket\rho(\lambda E'_1.\Psi(\underline{\underline{\mathtt{let}}}_i\ \Phi(E'_1)\ \underline{\underline{\mathtt{in}}} \\
& \quad \lambda v^\diamond.\mathcal{R}'_c\llbracket\Phi(E_2)[\ldots, v := v^\diamond]\rrbracket\kappa^\Phi))) \\
= \quad & \text{induction} \\
& \Phi(\mathcal{R}_c\llbracket E_1\rrbracket\rho(\lambda E'_1.\Psi(\underline{\underline{\mathtt{let}}}_i\ \Phi(E'_1)\ \underline{\underline{\mathtt{in}}} \\
& \quad \lambda v^\diamond.\Phi(\mathcal{R}_c\llbracket(E_2)\rrbracket\rho[v \mapsto v^\diamond]\kappa)))) \\
= \quad & \text{definition of } \Phi \text{ and cancellation } \Phi\Psi \\
& \Phi(\mathcal{R}_c\llbracket E_1\rrbracket\rho(\lambda E'_1.\underline{\mathtt{let}}_i\ v^\diamond = E'_1\ \underline{\mathtt{in}} \\
& \quad \mathcal{R}_c\llbracket(E_2)\rrbracket\rho[v \mapsto v^\diamond]\kappa)) \\
= \quad & \text{definition of } \mathcal{R}_c \\
& \Phi(\mathcal{R}_c\llbracket\underline{\mathtt{let}}_{i+1}\ v = E_1\ \underline{\mathtt{in}}\ E_2\rrbracket\rho\kappa)
\end{aligned}$$

The remaining cases are proved similarly.

B Proof of Lemma 3

We prove by induction on E:

$$\mathcal{R}'_c[\![E[v_i := E_i]]\!]\kappa = (E[v_i := \eta E_i])\kappa$$

case v_j

$$\begin{array}{rl} & \mathcal{R}'_c[\![\mathtt{var}\ v_j[v_i := E_i]]\!]\kappa \\ = & \text{substitution} \\ & \mathcal{R}'_c[\![\mathtt{var}\ E_j]\!]\kappa \\ = & \text{definition of } \mathcal{R}'_c \\ & \kappa E_j \\ = & \beta \text{ expansion} \\ & (\lambda\kappa.\kappa E_j)\kappa \\ = & \text{definition of } \eta \text{ and substitution} \\ & (v_j[v_i := \eta E_i])\kappa \end{array}$$

case $\underline{\underline{\lambda}}_0 \lambda v.E$

$$\begin{array}{rl} & \mathcal{R}'_c[\![(\underline{\underline{\lambda}}_0 \lambda v.E)[\ldots]]\!]\kappa \\ = & \text{definition of } \mathcal{R}'_c \\ & \kappa(\lambda y.\mathcal{R}'_c[\![((\lambda v.E)y)[\ldots]]\!]) \\ = & \eta \text{ expansion} \\ & \kappa(\lambda y.\lambda\kappa.\mathcal{R}'_c[\![((\lambda v.E)y)[\ldots]]\!]\kappa) \\ = & \beta \text{ reduction} \\ & \kappa(\lambda y.\lambda\kappa.\mathcal{R}'_c[\![E[\ldots, v := y]]\!]\kappa) \\ = & \text{induction} \\ & \kappa(\lambda y.\lambda\kappa.E[\ldots, v := \eta y]\kappa) \\ = & \beta \\ & \kappa(\lambda y.\lambda\kappa.((\lambda v.E)(\eta y)[\ldots])\kappa) \\ = & \eta \\ & \kappa(\lambda y.((\lambda v.E)(\eta y)[\ldots])) \\ = & \beta \\ & (\lambda\kappa.\kappa\lambda y.(\lambda v.E)(\eta y)[\ldots])\kappa \\ = & \text{definition of } \underline{\underline{\lambda}}_0 \\ & (\underline{\underline{\lambda}}_0(\lambda v.E))[\ldots]\kappa \end{array}$$

case $E_1 \underline{\underline{@}}_0 E_2$

$$\begin{array}{rl} & \mathcal{R}'_c[\![(E_1 \underline{\underline{@}}_0 E_2)[\ldots]]\!]\kappa \\ = & \text{substitution} \\ & \mathcal{R}'_c[\![(E_1[\ldots]) \underline{\underline{@}}_0 (E_2[\ldots])]\!]\kappa \\ = & \text{definition of } \mathcal{R}'_c \\ & \mathcal{R}'_c[\![E_1[\ldots]]\!](\lambda y_1.\mathcal{R}'_c[\![E_2[\ldots]]\!](\lambda y_2.y_1 y_2 \kappa)) \\ = & \text{induction} \\ & E_1[\ldots](\lambda y_1.E_2[\ldots](\lambda y_2.y_1 y_2 \kappa)) \\ = & \text{substitution and definition of } \underline{\underline{@}}_0 \\ & (\underline{\underline{@}}_0(E_1, E_2))[\ldots]\kappa \end{array}$$

case $\underline{\underline{\lambda}}_{i+1} \lambda v.E$

$$\begin{array}{rl} & \mathcal{R}'_c[\![(\underline{\underline{\lambda}}_{i+1} \lambda v.E)[\ldots]]\!]\kappa \\ = & \text{substitution} \\ & \mathcal{R}'_c[\![\underline{\underline{\lambda}}_{i+1}(\lambda v.E)[\ldots]]\!]\kappa \\ = & \text{definition of } \mathcal{R}'_c \\ & \kappa(\underline{\underline{\lambda}}_i \lambda v^\diamond.\mathcal{R}'_c[\![(\lambda v.E)[\ldots]v^\diamond]\!](\lambda z.z)) \\ = & \beta \\ & \kappa(\underline{\underline{\lambda}}_i \lambda v^\diamond.\mathcal{R}'_c[\![E[\ldots, v := v^\diamond]]\!](\lambda z.z)) \\ = & \text{induction} \\ & \kappa(\underline{\underline{\lambda}}_i \lambda v^\diamond.E[\ldots, v := \eta v^\diamond](\lambda z.z)) \\ = & \text{interpreting } \underline{\underline{\lambda}}_i \\ & \kappa\lceil\underline{\underline{\lambda}}_i \lambda v^\diamond.\langle E[\ldots, v := \eta v^\diamond](\lambda z.z)\rangle\rceil \\ = & \beta \\ & \kappa\lceil\underline{\underline{\lambda}}_i \lambda v^\diamond.\langle(\lambda v.E)[\ldots](\eta v^\diamond)(\lambda z.z)\rangle\rceil \\ = & \text{definition of } \underline{\underline{\lambda}}_{i+1} \\ & (\underline{\underline{\lambda}}_{i+1}(\lambda v.E)[\ldots])\kappa \\ = & \text{substitution} \\ & (\underline{\underline{\lambda}}_{i+1}(\lambda v.E))[\ldots]\kappa \end{array}$$

The remaining cases are proved similarly.

C Definition of shift and reset

The control operators **shift** and **reset** are defined through an extended CPS transformation [12] derived from Plotkin's call-by-value CPS transformation. **reset** executes its argument in a fresh context. **shift** captures (and abandons) the context up to the next lexically enclosing **reset** and abstracts it as a function.

$$\begin{array}{lcl} \mathcal{T}[\![v]\!] & = & \lambda\kappa.\kappa v \\ \mathcal{T}[\![\lambda v.e]\!] & = & \lambda\kappa.\kappa\lambda v.\mathcal{T}[\![e]\!] \\ \mathcal{T}[\![e_1\ e_2]\!] & = & \lambda\kappa.e_1\lambda v_1.e_2\lambda v_2.v_1 v_2 \kappa \\ \mathcal{T}[\![\mathbf{reset}\ e]\!] & = & \lambda\kappa.\kappa(\mathcal{T}[\![e]\!]\lambda z.z) \\ \mathcal{T}[\![\mathbf{shift}\ j.e]\!] & = & \lambda\kappa.\mathcal{T}[\![e]\!][j := \lambda v.\lambda\kappa'.\kappa'(\kappa v)](\lambda z.z) \end{array}$$