Two for the Price of One: Composing Partial Evaluation and Compilation

Michael Sperber   Peter Thiemann
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen
Sand 13, D-72076 Tübingen, Germany
{sperber,thiemann}@informatik.uni-tuebingen.de

Abstract

One of the flagship applications of partial evaluation is compilation and compiler generation. However, partial evaluation is usually expressed as a source-to-source transformation for high-level languages, whereas realistic compilers produce object code.

We close this gap by composing a partial evaluator with a compiler by automatic means. Our work is a successful application of several meta-computation techniques to build the system, both in theory and in practice. The composition is an application of deforestation or fusion.

The result is a run-time code generation system built from existing components. Its applications are numerous. For example, it allows the language designer to perform interpreter-based experiments with a source-to-source version of the partial evaluator before building a realistic compiler which generates object code automatically.

Keywords semantics-directed compiler generation, partial evaluation, compilation of higher-order functional languages, run-time code generationy

1 Introduction

Both partial evaluation and run-time code generation (RTCG) are program optimization techniques that have received considerable attention in the language implementation community. The literature describes partial evaluation as an optimizing program transformation based on constant propagation and memoization of generated procedures. Most partial evaluators perform a source-to-source transformation. In RTCG, on the other hand, the focus is on dynamically generating code fragments at run time and executing them.

Recently, researchers have begun to notice that partial evaluation terminology and techniques are suitable for specifying and implementing RTCG: Offline partial evaluation generates the output (or *residual*) program by stitching together pre-fabricated fragments of the source program, inserting constants in pre-determined places. Generic techniques for RTCG assemble pre-fabricated fragments of object code (usually called *templates*) in a similar manner [8, 9, 38, 39].

Since then, systems have been developed that implement RTCG with the help of partial evaluation tools [8, 9]. Notably, the *binding-time analysis*, which is a vital part of every offline partial evaluator, can automatically determine a proper staging of computations and thus guide the construction of the object code templates.

However, most implementations of partial evaluation systems work at the source level; with these systems, to obtain object code on-the-fly, it is necessary to employ reflection through `eval` procedures or similar means, thereby incurring significant compilation or interpretation overhead [58]. Partial evaluation systems which directly generate object code exist; however, their implementation usually involves hand-written code generation primitives [14, 26]. The implementation of many RTCG systems is complex and requires ad-hoc modifications to existing compilers or rewriting of components from scratch.

We describe how to obtain a portable partial evaluation system for Scheme [49] which directly generates object code in an essentially automatic way, by composing a partial evaluation system with a compiler. For the composition, no knowledge of the internal workings of the compiler (or even the nature of the output code) or the partial evaluator is necessary. We achieve the composition itself automatically through a specialization technique—*deforestation* [62]. Our methods allow for the automatic generation of run-time code generation systems.

Our system is modular by construction: It is possible to modify the partial evaluation system without keeping in mind object code generation issues, and to replace the compiler without any changes to the partial evaluation system or even "glue code" (which is generated automatically.) In fact, in our implementation, compiler and partial evaluation system were developed independently of each other.

As it turns out, the components of our system naturally complement each other: The partial evaluation system already generates code in a subset of Scheme especially suited for compilation and on-the-fly code generation. This allows us to use a simple compiler and to circumvent several analysis steps in the compiler of the underlying Scheme system.

To summarize, the contributions of our work are the following:

- We present a formalization of the specialization phase of an offline partial evaluator for a higher-order core language resembling Scheme.
- We show how to convert a recursive-descent compiler automatically into a set of code generation combinators.

- We show how to automatically compose the combinators with a partial evaluator using deforestation.
- We have implemented our techniques in the framework of an existing partial evaluation system [57] and *Scheme 48* [31], a byte-code implementation of Scheme.
- We have compiled realistic programs and run benchmarks that confirm the expected benefits of the system.

Our work has a number of immediate applications:

- For straightforward use in partial evaluation, our system avoids the compilation step for residual programs, thus speeding up the turnaround cycle in experimental applications of partial evaluation.
- The system facilitates the automatic construction of true compilers: It maps a language description (an interpreter) to a compiler that directly generates low-level object code.
- Our system allows the creation and execution of customized code at run time, thereby performing some "classic" jobs of RTCG systems.
- The system makes realistic *incremental specialization* feasible which not only allows for the implementation of dynamically evolving programs, but can also avoid termination problems in partial evaluation [58].

Since our implementation is based on a byte-code language system, we have not addressed native-code generation issues such as register allocation or code optimization. We believe these issues to be largely orthogonal to our technique.

Overview Because the implementation of our system was surprisingly simple, we give the history of the implementation process in Sec. 2. Section 3 gives some background on partial evaluation and program generator generators (PGG's). Section 4 introduces the concrete partial evaluation framework used in our current work. Section 5 describes the theoretical background of our work. In Sec. 6, we present the techniques we used in the implementation. Benchmarks are the subject of Sec. 7. Section 8, finally, gives an account of related work.

2 Run-Time Code Generation Made Easy

The script for our "RTCG production" turns out to be surprisingly simple.

Cast *Peter* and *Mike*, two programmers.

Setting Autumn 1996, a crowded university office in Tübingen, Germany, containing two desks with workstations.

Each of the workstations has installed a language system, in this case *Scheme 48* [31], and a partial evaluator [29], or, better, a *program generator generator* (PGG) [35,57], which works on the same language system.

The acts of the production correspond to the technical descriptions in Sections 6.1–6.4.

2.1 Act 1: Write a Compiler

Mike writes a simple compiler for the language system. The compiler needs to handle the output language of the partial evaluation process. As *Peter*, the author of the PGG can tell, that language happens to be in A-normal form (ANF), a small subset of Scheme especially suitable for effective compilation.

Mike chooses to chop down the stock Scheme 48 compiler to a compiler for programs in ANF and to introduce a few obvious optimizations possible through that. The new compiler uses exactly the same syntax dispatch and code generation conventions as the normal compiler, and is therefore seamlessly integrated with the base language system. In principle, this step can be avoided given a suitable ANF compiler.

2.2 Act 2: Annotate the Compiler

Peter obtains the compiler from *Mike* and decorates it with binding-time annotations. *Peter* resists learning about the internal workings of the compiler and the code-generation issues involved even though *Mike* assures him it is all very simple. However, *Peter* just needs to know *where* the compiler generates *code*—not how. *Peter* merely prefixes code-generating expressions in the compiler by an operator that causes them to be delayed until code-generation time. The syntax dispatch remains unchanged. Even when *Peter* is finished, he has no idea how the code works. Neither does *Mike*, poring over the code, understand the subtleties of *Peter*'s annotations.

2.3 Act 3: Implement the Annotations

Peter writes macros that will simply ignore the annotations, so the result is still usable as an ordinary compiler. He writes a second set of macros which turn the compiler functions into combinators. These combinators will replace counterparts in the PGG normally responsible for producing output code in the source language. The new combinators directly produce object code. Both jobs turn out to be reasonably straightforward.

2.4 Act 4: Test and Debug

Peter and *Mike* get together at a terminal to hook up the PGG with the new code generation combinators. Tension is high because the components have been developed separately, have not been tried in combination, debugging support at the object code level is minimal, and the paper deadline is approaching. Will they make it?

Right off, *Peter* and *Mike* find a number of embarrassing bugs in the binding-time analysis of the PGG, and even more embarrassing bugs in *Mike*'s original compiler. They also have to resolve some integration issues that have to do with the fact that the new program generators manipulate object code generators rather than source expressions.

To their surprise and slight disappointment, there are no conceptual problems, and no hackery whatsoever is necessary to ensure that the compiler is usable both as a normal source code compiler and as a generator for the code generation combinators.

Curtain

$$
\begin{array}{rcl}
M & ::= & V \\
& | & (\textbf{let}\ (x\ M_1)\ M_2) \\
& | & (\textbf{if0}\ M_1\ M_2\ M_3) \\
& | & (\textbf{@}\ M_0\ M_1 \ldots M_n) \\
& | & (O\ M_1 \ldots M_n) \\
\\
V & ::= & c \mid x \mid (\lambda x_1 \ldots x_n.M)
\end{array}
\qquad
\begin{array}{rcl}
V & \in & \textsf{Values} \\
c & \in & \textsf{Constants} \\
x & \in & \textsf{Variables} \\
O & \in & \textsf{Primitives}
\end{array}
$$

Figure 1: Abstract syntax of Core Scheme expressions (CS)

$$
\begin{array}{rcl}
M & ::= & V \\
& | & (\textbf{let}\ (x\ V)\ M) \\
& | & (\textbf{if0}\ V\ M_1\ M_2) \\
& | & (\textbf{@}\ V\ V_1 \ldots V_n) \\
& | & (\textbf{let}\ (x\ (\textbf{@}\ V\ V_1 \ldots V_n))\ M) \\
& | & (O\ V_1 \ldots V_n) \\
& | & (\textbf{let}\ (x\ (O\ V_1 \ldots V_n))\ M) \\
\\
V & ::= & c \mid x \mid (\lambda x_1 \ldots x_n.M)
\end{array}
$$

Figure 2: Syntax of CS terms in A-normal form

3 Partial Evaluation and PGG's

Partial evaluation [7, 29] is an automatic program specialization technique. It operates on a source program and its known (*static*) input. The specialized *residual program* takes the remaining (*dynamic*) input and delivers the same result as the source program applied to the whole input. Often, the residual program is faster than the source program.

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

$$
\begin{array}{llcll}
[\![\texttt{p-gen}]\!] & \texttt{s-inp} & & = & \texttt{p}_{\texttt{s-inp}} \\
[\![\texttt{p}_{\texttt{s-inp}}]\!] & & \texttt{d-inp} & = & \texttt{result} \\
[\![\texttt{p}]\!] & \texttt{s-inp} & \texttt{d-inp} & = & \texttt{result}
\end{array}
$$

A generating extension results from applying a program-generator generator (PGG) to `p`. A PGG can result from double self-application of a partial evaluator as described by the third Futamura projection [20, 29, 61], or from direct implementation [35, 57].

4 Specializing Core Scheme

The building blocks of the system are a PGG for a large subset of Scheme [57] and the Scheme 48 byte code implementation of Scheme [31]. Our PGG has several design properties which make it especially suitable for our goals. None of these properties were designed with object-code generation in mind—they arise naturally from other requirements.

For our presentation, we use an abstract syntax for "Core Scheme" (CS) [19] shown in Fig. 1. We have omitted top-level definitions for brevity's sake.

During work on a partial evaluator for an ML-style language that performs side effects at specialization time [15], Dussart, Lawall, and the second author discovered that a specializer must output code in a restricted subset of Scheme that explicitly serializes computations to ensure correctness—essentially A-normal form [19] (ANF) as shown in Fig. 2. Thus, ANF is the natural target language of the PGG.

Figure 3 shows such a specializer restricted to Core Scheme. In the definition of the specializer, we employ ACS ("Annotated Core Scheme"). ACS has additional variants of primitive operations, **let** expressions, lambda abstractions, applications, and conditionals (annotated with superscript ^{D}, for dynamic) that generate code. Additionally, there is a **lift** construct that coerces first-order values to code. Underlining indicates code generation. The superscript $^\diamond$ produces fresh variables. Multiple occurrences of, say, $x^\diamond$ denote the same variable.

The specializer employs *continuation-based partial evaluation* [2, 6, 37] to generate code in ANF. Whenever a piece of code denoting a "serious" computation (a non-value) is constructed, it is wrapped in a **let** expression with a fresh variable which is used in place of the piece of code. This happens in the rules for primitive operations and applications. The **let** wrapping is not necessary for values (constants, variables, and abstractions).

The specializer performs some other transformations that are necessary in a compiler. It desugars input programs to Core Scheme, performs lambda lifting [28] and assignment elimination. We do not show the parts of the specializer that deal with memoization, since they are standard [29, 58] and not relevant to the present work.

Our compiler makes essential use of the transformations already done by the specializer. It is a straightforward recursive-descent compiler which passes around source expressions, a compile-time environment mapping names to stack and environment locations, and a stack depth necessary to correctly generate code for the Scheme 48 virtual machine.

$$
\begin{array}{lcl}
\mathcal{S}[\![c]\!]\rho & = & \lambda k.kc \\
\mathcal{S}[\![x]\!]\rho & = & \lambda k.k(\rho[\![x]\!]) \\
\mathcal{S}[\![(O\ E_1 \ldots E_n)]\!]\rho & = & \lambda k.\mathcal{S}[\![E_1]\!]\rho(\lambda y_1.\ldots\mathcal{S}[\![E_n]\!]\rho(\lambda y_n.k([\![O]\!]y_1 \ldots y_n))) \\
\mathcal{S}[\![(\lambda x_1 \ldots x_n.E)]\!]\rho & = & \lambda k.(\lambda y_1 \ldots y_n.\mathcal{S}[\![E]\!]\rho[x_i/y_i]) \\
\mathcal{S}[\![(@\ E_0\ E_1 \ldots E_n)]\!]\rho & = & \lambda k.\mathcal{S}[\![E_0]\!]\rho(\lambda f.\mathcal{S}[\![E_1]\!]\rho(\lambda y_1.\ldots\mathcal{S}[\![E_n]\!]\rho(\lambda y_n.fy_1 \ldots y_n k)) \\
\mathcal{S}[\![(\mathbf{let}\ (x\ E_1)\ E_2)]\!]\rho & = & \lambda k.\mathcal{S}[\![E_1]\!]\rho(\lambda y.\mathcal{S}[\![E_2]\!]\rho[x/y]k) \\
\mathcal{S}[\![(\mathbf{if0}\ E_1\ E_2\ E_3)]\!]\rho & = & \lambda k.\mathcal{S}[\![E_1]\!]\rho(\lambda y.(\mathbf{if0}\ y\ (\mathcal{S}[\![E_2]\!]\rho k)\ (\mathcal{S}[\![E_3]\!]\rho k))) \\
\\
\mathcal{S}[\![(\mathbf{lift}\ E)]\!]\rho & = & \lambda k.\mathcal{S}[\![E]\!]\rho(\lambda y.k(\underline{y})) \\
\mathcal{S}[\![(O^D\ E_1 \ldots E_n)]\!]\rho & = & \lambda k.\mathcal{S}[\![E_1]\!]\rho(\lambda y_1.\ldots\mathcal{S}[\![E_n]\!]\rho(\lambda y_n.(\underline{\mathbf{let}}\ (x^\diamond\ (\underline{O}\ y_1 \ldots y_n))\ kx^\diamond))) \\
\mathcal{S}[\![(\lambda^D x_1 \ldots x_n.E)]\!]\rho & = & \lambda k.k((\underline{\lambda} x_1^\diamond \ldots x_n^\diamond.\mathcal{S}[\![E]\!]\rho[x_i^\diamond/x_i](\lambda y.y))) \\
\mathcal{S}[\![(@^D\ E_0\ E_1 \ldots E_n)]\!]\rho & = & \lambda k.\mathcal{S}[\![E_0]\!]\rho(\lambda y.\mathcal{S}[\![E_1]\!]\rho(\lambda y_1.\ldots\mathcal{S}[\![E_n]\!]\rho(\lambda y_n.(\underline{\mathbf{let}}\ (x^\diamond\ (\underline{@}\ y\ y_1 \ldots y_n))\ kx^\diamond))) \\
\mathcal{S}[\![(\mathbf{let}^D\ (x\ E_1)\ E_2)]\!]\rho & = & \lambda k.\mathcal{S}[\![E_1]\!]\rho(\lambda y.\mathcal{S}[\![E_2]\!]\rho[y/x]k) \\
\mathcal{S}[\![(\mathbf{if0}^D\ E_1\ E_2\ E_3)]\!]\rho & = & \lambda k.\mathcal{S}[\![E_1]\!]\rho(\lambda y_1.(\underline{\mathbf{if0}}\ y_1\ (\mathcal{S}[\![E_2]\!]\rho k)\ (\mathcal{S}[\![E_3]\!]\rho k)))
\end{array}
$$

Figure 3: Specializer which generates ANF output code

5 Automatic Composition

For the theoretical basis of our work, we draw from ideas from numerous disciplines: Algebraic syntax representation, compositional compilers, and deforestation are the key ideas crucial to composing a specializer with a compiler. Moreover, ANF serves as a convenient means of communication between the specializer and the compiler.

5.1 Syntax

For our purposes we regard standard syntax and annotated syntax as algebraic datatypes. Therefore, both can be defined as least fixpoints of functions over sets of expressions. The functions are derived from the grammar of CS (with $+$ denoting disjoint union, $\times$ denoting cartesian product of sets, and $\mathsf{List}(X)$ denoting the set of finite lists over X). Figure 4 shows the definition. We use the symbolic tags

$$\mathsf{Syntax} \quad = \quad \mathsf{MkSyntax}(\mathsf{Syntax})$$

where

$$
\begin{array}{lll}
\multicolumn{2}{l}{\mathsf{MkSyntax}(X)} & \\
= & \mathit{const}\ \mathsf{Constants} & \text{constants} \\
+ & \mathit{var}\ \mathsf{Variables} & \text{identifiers} \\
+ & \mathit{lam}\ (\mathsf{List}(\mathsf{Variables}) \times X) & \text{lambda abstractions} \\
+ & \mathit{let}\ (\mathsf{Variables} \times X \times X) & \text{let expressions} \\
+ & \mathit{if}\ (X \times X \times X) & \text{conditionals} \\
+ & \mathit{app}\ (X \times \mathsf{List}(X)) & \text{applications} \\
+ & \mathit{prim}\ (\mathsf{Primitives} \times \mathsf{List}(X)) & \text{primitive operations}
\end{array}
$$

Figure 4: Algebraic Definition of Syntax

const, *var*, *lam*, *if*, *app*, and *prim* as indicators in which summand of the disjoint union a value in $\mathsf{MkSyntax}(X)$ lies. This allows us to use pattern matching as syntactic sugar in defining functions on $\mathsf{MkSyntax}(X)$.

Given MkSyntax, we can convert every function $f : Y \to Z$ to a function $\mathsf{MkSyntax}(f) : \mathsf{MkSyntax}(Y) \to \mathsf{MkSyntax}(Z)$ by

$$
\begin{array}{lcl}
\mathsf{MkSyntax}(f)(\mathit{const}\ c) & = & \mathit{const}\ c \\
\mathsf{MkSyntax}(f)(\mathit{var}\ (x)) & = & \mathit{var}\ (x) \\
\mathsf{MkSyntax}(f)(\mathit{lam}\ (x_1 \ldots x_n, y)) & = & \mathit{lam}\ (x_1 \ldots x_n, fy) \\
\mathsf{MkSyntax}(f)(\mathit{let}\ (x, y_1, y_2)) & = & \mathit{let}\ (x, fy_1, fy_2) \\
\mathsf{MkSyntax}(f)(\mathit{if}\ (y_1, y_2, y_3)) & = & \mathit{if}\ (fy_1, fy_2, fy_3) \\
\mathsf{MkSyntax}(f)(\mathit{app}\ (y, y_1 \ldots y_n)) & = & \mathit{app}\ (fy, fy_1 \ldots fy_n) \\
\mathsf{MkSyntax}(f)(\mathit{prim}\ (O, y_1 \ldots y_n)) & = & \mathit{prim}\ (O, fy_1 \ldots fy_n)
\end{array}
$$

Analogously, we define

$$\mathsf{AnnSyntax} = \mathsf{MkAnnSyntax}(\mathsf{AnnSyntax}),$$

the function MkAnnSyntax, and its action on functions $f : Y \to Z$. The additional tags are *lift*, *dlam*, *dif*, *dapp*, and *dprim*.

Technically, MkSyntax and MkAnnSyntax are functors over Set.

5.2 Compositionality

One of the fundamental ideas of denotational semantics [51] is the description of the meaning of a programming language phrase by a *compositional* recursive definition: The meaning of an expression is a function of the meanings of its subexpressions.

For the language CS we can therefore describe the semantics of CS by defining suitable domains and functions *ev-const*, *ev-var*, *ev-lam*, *ev-let*, *ev-if*, *ev-app*, and *ev-prim*. (A *denotational implementation* to follow Espinosa's terminology [18, p. 11].) They are parameters to a generic recursion schema that traverses CS expressions (see Fig. 5) where we write $\overline{ev}$ for the tuple (*ev-const*, ..., *ev-prim*). This recursion schema is a *catamorphism* for Syntax [43].

Apart from compositional semantics, catamorphisms are also useful for describing compilers and specializers [57]. For a compiler, the functions $\overline{ev}_C$ are compilation functions for each single construct. For a specializer, the functions $\overline{ev}_S$ are specialization functions. In this case we have to use an extended schema $\mathit{cata}_{ACS}(\overline{ev}_S)(\_)$ which has additional parameters and cases for the annotated versions of the syntax constructors. For example the specialization function for $(\mathbf{if0}^D\ M_1\ M_2\ M_3)$ is

$$\mathit{ev\text{-}dif}_S(z_1, z_2, z_3) = \lambda\rho.\lambda k.z_1\rho(\lambda y_1.(\underline{\mathbf{if0}}\ y_1\ (z_2\rho k)\ (z_3\rho k))).$$

It is obtained from the explicit recursive definition in Fig. 3 by systematic transformation.

$$
\begin{array}{lcl@{\quad}c@{\quad}l}
cata_{CS}(\overline{ev})(M) & = & \texttt{case}\ M\ \texttt{of} & & \\
 & & c & \Rightarrow & ev\text{-}const(c) \\
 & & v & \Rightarrow & ev\text{-}var(v) \\
 & & (\lambda x_1 \ldots x_n.M) & \Rightarrow & ev\text{-}lam(x_1 \ldots x_n, cata_{CS}(\overline{ev})(M)) \\
 & & (\textbf{let}\ (x\ M_1)\ M_2) & \Rightarrow & ev\text{-}let(x, cata_{CS}(\overline{ev})(M_1), cata_{CS}(\overline{ev})(M_2)) \\
 & & (\textbf{if0}\ M_1\ M_2\ M_3) & \Rightarrow & ev\text{-}if(cata_{CS}(\overline{ev})(M_1), cata_{CS}(\overline{ev})(M_2), cata_{CS}(\overline{ev})(M_3)) \\
 & & (\textbf{@}\ M\ M_1 \ldots M_n) & \Rightarrow & ev\text{-}app(cata_{CS}(\overline{ev})(M), cata_{CS}(\overline{ev})(M_1) \ldots cata_{CS}(\overline{ev})(M_n)) \\
 & & (O\ M_1 \ldots M_n) & \Rightarrow & ev\text{-}prim(O, cata_{CS}(\overline{ev})(M_1) \ldots cata_{CS}(\overline{ev})(M_n))
\end{array}
$$

Figure 5: Generic recursion schema for CS

5.3 From Recursive Definition to Implicit Recursion

The transformation of recursive definitions into definitions using catamorphisms is non-trivial in general [36]. In our specific case, standard techniques from partial evaluation [29] suffice to transform the explicit recursive definition of the compiler into the specialized compilation functions $\overline{ev}_C$. Our aim is to generate $\overline{ev}_C$ automatically from the given recursive definition by specializing it with respect to the different syntactic constructs. Two things have to be done:

1. The syntactic dispatch has to be performed at specialization time.
2. The recursive calls to the (explicitly recursive) compilation function on the syntactic subcomponents of the construct have to be removed (replaced by the identity).

For the removal of recursive calls, the definition of $cata_{CS}(\overline{ev})(\_)$ already takes care of the recursion. Hence all syntactic subcomponents have already been compiled when $\overline{ev}_C$ is applied. For the specialization combinators $\overline{ev}_S$, such a transformation has been carried out and proved correct by the second author [57].

From now on we assume that the compiler is given in the form $cata_{CS}(\overline{ev}_C)(\_)$ and the specializer in the form $cata_{ACS}(\overline{ev}_S)(\_)$. The types of these functions are interesting and important for us.

$$
\begin{array}{lcl}
\overline{ev}_C & : & \mathsf{MkSyntax}(\mathsf{Code}) \to \mathsf{Code} \\
cata_{CS}(\overline{ev}_C)(\_) & : & \mathsf{Syntax} \to \mathsf{Code} \\
\overline{ev}_S & : & \mathsf{MkAnnSyntax}(\mathsf{Syntax}) \to \mathsf{Syntax} \\
cata_{ACS}(\overline{ev}_S)(\_) & : & \mathsf{AnnSyntax} \to \mathsf{Syntax}
\end{array}
$$

5.4 Deforestation

Deforestation is a program transformation for functional programs that removes intermediate data structures by symbolic composition [62]. Suppose a function g produces some intermediate data structures that is immediately consumed by function f in $f(g(x))$. Deforestation (if applicable) results in a function h such that $\forall x.h(x) = f(g(x))$ where the computation of h does not involve the construction of intermediate data. Deforestation can also be expressed in a calculational form where it amounts to the application of a so-called *fusion* or *promotion* theorem [41, 43]. Specialized to our situation, the fusion theorem states the following:

$$
\begin{array}{ll}
\forall X.\forall \overline{y}. & cata_{CS}(\overline{ev}_C)(ev\text{-}X_S(\overline{y})) \\
 & = ev\text{-}X_{C\circ S}(\mathsf{MkAnnSyntax}(cata_{CS}(\overline{ev}_C)(\_))(\overline{y})) \\
\Rightarrow & \\
\forall M. & cata_{CS}(\overline{ev}_C)(cata_{ACS}(\overline{ev}_S)(M)) \\
 & = cata_{ACS}(\overline{ev}_{C\circ S})(M)
\end{array}
$$

Here, X ranges over the syntax constructor tags of annotated expressions and $\overline{y}$ is an argument vector for X, but with all arguments of type $\mathsf{AnnSyntax}$ replaced by arguments of type Syntax, i.e., $X(\overline{y}) : \mathsf{MkAnnSyntax}(\mathsf{Syntax})$. That is, on the left side of the premise the specializer has recursively specialized $\mathsf{AnnSyntax}$ to Syntax and is now about to specialize the next constructor of $\mathsf{AnnSyntax}$. Now, $ev\text{-}X_S(\overline{y}) : \mathsf{MkSyntax}(\mathsf{Syntax})$ performs this specialization step and we can apply $cata_{CS}(\overline{ev}_C)(\_) = \lambda z.cata_{CS}(\overline{ev}_C)(z)$ to it to compile it to Code.

On the right side of the premise we use $ev\text{-}X_{C\circ S}$, a function that specializes and compiles the annotated construct X. Here, we first compile the components of $\overline{y}$ of type Syntax to obtain a value of type $\mathsf{MkAnnSyntax}(\mathsf{Code})$. This is a suitable argument for $ev\text{-}X_{C\circ S}$ since

$$
\begin{array}{lcl}
\overline{ev}_{C\circ S} & : & \mathsf{MkAnnSyntax}(\mathsf{Code}) \to \mathsf{Code} \\
cata_{ACS}(\overline{ev}_{C\circ S})(\_) & : & \mathsf{AnnSyntax} \to \mathsf{Code}
\end{array}
$$

The remaining M in the conclusion ranges over $\mathsf{AnnSyntax}$.

Now the task is: Given $\overline{ev}_S$ and $\overline{ev}_C$ find $\overline{ev}_{C\circ S}$ such that the premise of the fusion theorem holds. In this way we can derive that, for example,

$$ev\text{-}dif_{C\circ S}(z_1, z_2, z_3) = \lambda\rho.\lambda k.z_1\rho(\lambda y_1.ev\text{-}if_C(y_1, z_2\rho k, z_3\rho k)).$$

Now we know what we have to do to obtain $\overline{ev}_{C\circ S}$: we only have to replace the syntax constructor X in the definition of $\overline{ev}_S$ by the respective call to function $ev\text{-}X_C$ from $\overline{ev}_C$. In practice, we parameterize $\overline{ev}_S$ over the (standard) syntax constructors and provide alternative implementations for them: one that constructs syntax and another one that corresponds to $\overline{ev}_C$.

6 Implementation

The theory presented in the previous section translates into practice smoothly. This section describes the concrete implementation of the compiler and its fusion with the PGG in the context of the Scheme 48 system.

6.1 Step 1: Write a Compiler

In principle, it is possible to simply use the stock Scheme 48 byte-code compiler which passes a compile-time continuation to identify tail-calls. However, the target code of the specialization engine is in ANF. ANF, as shown in Fig. 2, already makes control flow explicit. Only those function applications wrapped in a **let** are non-tail calls; all others are jumps. Hence, the propagation of a compile-time continuation is unnecessary, and it is sensible to make do with a drastically cut-down version of the compiler. Removing the compile-time continuation simplifies the compiler, and also speeds up later code generation, as it could not be removed by fusion.

The compiler is integrated with the Scheme 48 system. In particular, it uses its native syntax representation and dispatch mechanism. The output of the compiler is an abstract representation of the byte code for the Scheme 48 virtual machine, essentially a stack machine with direct support for closures and continuations [31].

Here is the compilator for `if`:

```
(define-compilator 'if syntax-type
  (lambda (node cenv depth)
    (let ((exp (node-form node))
          (alt-label (make-label)))
      (sequentially
       ;; Test
       (compile-trivial (cadr exp) cenv)
       (instruction-using-label
         (enum op jump-if-false)
         alt-label)
       ;; Consequent
       (compile (caddr exp) cenv depth)
       ;; Alternative
       (attach-label
         alt-label
         (compile (cadddr exp) cenv depth))))))
```

The compilator takes three parameters:

`node` a node of a syntax tree representing a conditional,

`cenv` a compile-time environment, and

`depth` the current depth of the stack.

`Compile` and `compile-trivial` compile the subexpressions of the conditional.

A compilator constructs object code by using a number of constructors: `Sequentially` arranges byte-code instructions in sequence; `make-label`, `instruction-using-label`, and `attach-label` serve to create the jump code typical for compiling conditionals. These constructors return an abstract representation of object code. Scheme 48 internally relocates the representation, resolves labels, and generates the actual byte code. The relocation step is inherent in the Scheme 48 architecture; an alternative implementation would generate the object code directly, using backpatching for resolving labels.

The compiler utilizes the `define-compilator` procedure to create an entry in a syntax dispatch table `compilators`. It uses the native syntax dispatch mechanism of the Scheme 48 compiler:

```
(define (define-compilator name type proc)
  (operator-define! compilators name type proc))
```

`Compilators` is then used by a procedure `compile` which is the top-level dispatcher of the compiler.

```
(define (compile exp cenv depth)
  (let ((node (classify exp cenv)))
    ((operator-table-ref compilators
                         (node-operator-id node))
     node cenv depth)))
```

`Compile` compiles serious expressions. An analogous mechanism creates the `compile-trivial` procedure.

6.2 Step 2: Annotate the Compiler

The annotation of the compiler functions is a straightforward process that requires no deep understanding of the code. The compilation combinators have to perform the syntactic dispatch at generation time (of the combinators) and copy the remaining code verbatim. The annotated compilator for `if` is as follows:

```
(define-compilator 'if syntax-type
  (lambda (node cenv depth)
    (let ((exp (node-form node)))
      (_let ((alt-label (_ make-label)))
        (_ sequentially
           ;; Test
           (compile-trivial (cadr exp) cenv)
           (_ instruction-using-label
              (_lift-literal
                 (enum op jump-if-false))
              alt-label)
           ;; Consequent
           (compile (caddr exp) cenv depth)
           ;; Alternative
           (_ attach-label alt-label
              (compile (cadddr exp) cenv depth)))))))
```

In an annotated program all constructs starting with an underline `_` perform code generation. The remaining parts are executed at generation time.

Of the parameters to the compilator, only the structure of `node` is known at generation time. The subexpressions of `node` as well as `cenv` and `depth` are unknown. A correct annotation prescribes code generation for every value that depends on an unknown value.

The underlined constructs have the following meaning:

- `_let` generates a `let` expression;
- `_` generates a procedure call, for example, `(_ make-label)` generates a call to procedure `make-label`;
- `_lift-literal` turns a generation-time constant into code.

Conceivably, even this annotation could have been performed automatically by an appropriate binding-time analysis, possibly at the expense of changing the representation of abstract syntax in the compiler.

6.3 Step 3: Implement the Annotations

Since the annotated compiler needs to serve both as a standalone compiler and as a generator for the code generation combinators, there are two different "implementations" of the annotations.

6.3.1 Annotations for the Compiler

For the compiler, the annotations need to "disappear" again. Scheme macros [49] do the job:

```
(define-syntax _
  (syntax-rules ()
    ((_ arg ...)
     (arg ...))))

(define-syntax _let
  (syntax-rules ()
    ((_let stuff ...)
```

```
    (let stuff ...))))

(define-syntax _lift-literal
  (syntax-rules ()
    ((_lift-literal z)
     z)))
```

6.3.2 Code Generation Combinators

Producing code generation combinators from the compiler is also straightforward. The idea is to create alternative versions of the annotation macros that produce Scheme source expressions for the combinators, to print these into a file, and load them when needed.

The macro `_` constructs a function call by taking the name `fct` of the function literally and processing the arguments recursively.

```
(define-syntax _
  (syntax-rules ()
    ((_ fct arg ...)
     `(fct ,arg ...))))
```

The `_let` macro constructs a **let** expression from the variable name `v` and the processed header `e`. The `body` is constructed in an environment where `v` is bound to the symbol `v`. This binding is produced by a generation-time **let** expression.

```
(define-syntax _let
  (syntax-rules ()
    ((_let ((v e)) body)
     `(LET ((v ,e))
        ,(let ((v 'v)) body)))))
```

In general, such a macro cannot reuse the variable name `v` but rather needs to generate fresh names [57]. This is not necessary in our implementation.

The `_lift-literal` macro (for numbers, etc) is merely present for conceptual reasons as numbers, for instance, are self-quoting in Scheme. To generate code for constants like lists, another macro `_lift-quote` inserts the proper quoting.

```
(define-syntax _lift-literal
  (syntax-rules ()
    ((_lift-literal z) z)))
```

The calls to `compile` and `compile-trivial` are discarded as explained in Sec. 5.4.

```
(define-syntax compile
  (syntax-rules ()
    ((compile arg ...)
     `(,arg ...))))
```

Finally, an alternative implementation of `define-compilator` generates procedure definitions for output code constructors. The generating extensions produced by the PGG call the procedures to generate residual code.

```
(define-syntax define-compilator
  (syntax-rules
    (quote if call
     let-trivial let-serious
     begin)
    ((define-compilator 'if _ ncd-fun)
     `(define (make-residual-if test then alt)
        ,(construct-serious-body
          ncd-fun
          (make-node (get-operator 'if)
                     '(IF TEST THEN ALT)))))
    ((define-compilator 'call _ ncd-fun)
     `(define (make-residual-call f . args)
        ,(construct-serious-body
          ncd-fun
          (make-node (get-operator 'call)
                     '(F . ARGS)))))
    ;; ...
))

(define (construct-serious-body ncd-fun node)
  `(lambda (cenv depth)
     ,(ncd-fun node 'cenv 'depth)))
```

Each of the compilators calls `construct-serious-body`. It accepts a function of three arguments, a known `node`, unknown `cenv`, and `depth` as described above. The other argument `node` is statically constructed in the respective part of the `define-compilator` macro.

The `make-residual-...` functions generated by the above process serve as direct replacements of the syntax construction functions (<u>**if0**</u> _ _ _), (<u>**@**</u> _ _), etc, in the specializer.

6.4 Step 4: Test and Debug

There were no problems in putting the system together. During the integration we discovered bugs mostly in the specializer and in the compiler which were independent of the conceptual issues.

The only problem to be resolved has to do with the duality between variable names and their compilators:

During ordinary specialization there are two kinds of objects: static values and pieces of code. Naive application of our approach replaces the pieces of code by compilation functions. However, the compilator for lambda abstractions requires a list of the *names* of its free variables; references to them are compiled differently from regular variables for the Scheme 48 VM. Therefore, our systems passes the names of variables by default and converts them to compilation functions when necessary.

7 Benchmarks

Our experiments largely confirm the expectations to RTCG technology, but also point to some possible improvements to our implementation.

For our benchmarks, we used two standard examples for compilation by partial evaluation: an interpreter for a small first-order functional language called Mixwell, and one for a small lazy functional language called Lazy, both taken from the Similix distribution [3]. The Mixwell interpreter is 93 lines long and was run on a 620-line input program, the Lazy interpreter has 127 lines of code and was run on a 26-line input program. We used Scheme 48 0.46 on a Pentium/90 laptop with 24 Megabytes of RAM running FreeBSD 2.1.5. All timings are cumulative over a large number of runs, and are in seconds.

	source code	object code
Mixwell	3.072	3.770
Lazy	1.832	3.451

Figure 6: Generation speed

Figure 6 shows timings for generating both Scheme source and object code directly for compilers generated from the interpreters, in both cases on medium-sized input programs. Object code generation is up to a factor of 2 slower than generating source, since Scheme 48 uses a higher-order representation for the object code that still needs to be converted to actual byte codes—that conversion is also part of the timings. Hence, a future step would be emitting byte code directly or using a more efficient intermediate representation.

MIXWELL	7.180
LAZY	4.746

Figure 7: Compilation times for the specialization output

Still, loading the generated source code back into the Scheme system is by far more expensive than direct object code generation, as in Fig. 7. Here, we used our own ANF compiler, not the (slower) stock Scheme 48 compiler. To fully appreciate the timing data, note that in order to produce object code for a specialized program from an ordinary specializer, we have to add the timings for source code generation in Fig. 6 and the compilation times in Fig. 7.

	BTA	Load	Generate	Compile
MIXWELL	2.730	4.026	0.652	0.964
LAZY	2.253	3.217	0.568	0.604

Figure 8: Using RTCG for normal compilation

One of our future objectives is to create a Scheme system where the stock compiler works through run-time code generation. For "normal" compilation, the system takes all inputs to a program as dynamic. Figure 8 shows timings for preliminary experiments in that direction: The "BTA" column shows the time needed for binding-time analysis and creation of the object code generator, "Load" is the time needed for loading (and compiling) the object code generator, and "Generate" the time for running it. "Compile" is the time needed to load and compile the original interpreter using the stock Scheme 48 compiler.

8 Related Work

8.1 A-Normal Form

Compilation with ANF [19] captures the essence of continuation-based compilation [1,30,33,55]. We build upon that work to construct the simple ANF compiler. Using ANF (or monadic normal form) for compilation is also put forward by Hatcliff and Danvy [24] and by Sabry and Wadler [50].

Danvy's work [10,11,13] uses type-directed partial evaluation for semantics-directed compilation. His system wraps **let** expressions around code that denotes computations in order to avoid code duplication. As a result, he also obtains programs in ANF. The type-directed partial evaluator is also a suitable candidate for composition with a compiler in the same way as shown in this work.

Partial evaluation [7,29] is an automatic program transformation that specializes programs with respect to parts of the input which are known in advance. Continuation-based partial evaluation [2, 37] is the enabling technology that makes our specializer suited to generate code in ANF. The partial evaluator that we use is the ANF version of Consel and Danvy's [6] initial approach to improve the results of partial evaluation by CPS transformation. The original application of our specializer is specialization of ML-style programs which can perform operations on references at specialization time [59].

8.2 Partial Evaluation

Holst [26] describes a system called AMIX, a partial evaluator that generates code for a stack machine directly. The motivation behind this system is similar to ours, with two notable differences: The AMIX system was written from scratch with the generation of stack code in mind and it is offline in the sense that it produced stack code in some representation that had to be fed to a separate interpreter. In contrast, our system results from the systematic composition of existing parts and it produces code for immediate execution by the run-time system.

Annotation functions similar to the ones shown in Sec. 2.3 are considered in work on writing PGG's by hand [22, 57]. Deforestation is well-known and well-investigated in the functional programming community because it is a powerful tool for program optimization [4,5,21,23,36,44,45,52–54,56,62].

Symbolic composition is an important technique in the *CERES* compiler generator system [60]. One of the key steps in *CERES* is the composition of a language definition considered as a compiler with a fixed compiler to a low-level language. However, the technical details of this composition are not spelled out. Also, in *CERES*, the result of the composition is the entire compiler, whereas our composition generates bricks from which the generated compiler will be built. Another application of composition in *CERES* is the creation of parts of the system itself.

8.3 Run-Time Code Generation

Run-time code generation has received revived interest since the early 90's when techniques became available to perform RTCG cheaply. Previously, RTCG happened mainly in the context of reflective language systems, notably Lisp, and usually involved prohibitive interpretive overhead or compilation time. Since then, it has been applied to code optimization [32], efficient dynamic implementation of programming languages [25], optimization of bitmap graphics [14,47], operating system optimization [42] and other tasks specifically suited to RTCG [8,9,38,39]. These present only selections; a complete overview of the field would exceed the scope of this paper.

Of particular interest is the work relating RTCG to partial evaluation. The FABIUS system of Lee and Leone [38,39] performs RTCG of native code for ML. FABIUS comes with its own application specific code generator, whereas we have reused the Scheme 48 code generation machinery. Both systems compile annotated source code to yield a program that generates code at run-time. The systems differ in the way they generate code. FABIUS produces object code at run-time and performs various standard optimizations at compile-time and at run-time. For example, it resolves the issue of register allocation for run-time generated code at compile-time whereas it addresses instruction selection at run-time. Our system abstracts from all these issues since the emphasis is on the composition itself. Our approach is complementary in that it could benefit from FABIUS's techniques to improve the efficiency and quality of code generation.

Consel's group has implemented the *Tempo* system for C [8, 9] which generates code templates which are copied and instantiated at run time. Their system interacts with the GNU C compiler, but requires low-level modifications in the code generator to support templates. Draves [14] has applied partial evaluation to low-level code, and implemented a PGG for an intermediate representation to obtain efficient code for computer graphics. His system operates purely on the intermediate code; only rudimentary support for higher-level programming is provided.

9 Assessment and Future Work

Unfortunately, researchers have applied the term "run-time code generation" to a variety of different situations. There is no clear cut border between ordinary compilation and RTCG. On the "ordinary" end of the spectrum, compilation is *offline* as in traditional compilers. On the other end of the spectrum, there are *online* systems like DCG [17] or the Synthesis kernel [42] that compile code for immediate execution. The rest of the spectrum is not empty: programming languages like ML, Scheme, or Smalltalk have a read-eval-print loop that accepts function definitions that are compiled and the code is immediately available for execution. Hence, they are essentially online compilers. Obviously, the trade-offs involved at each point of the spectrum are different.

Typical situations for RTCG seem to be the interleaving of compilation with execution of the compiled code, the generation of object code without explicitly running a compiler, and combinations of those two. Recent applications have focused on using RTCG for on-the-fly code optimizations in high-performance systems by staging computations at run time. For this to be effective, it is clearly necessary to deal with native code generation and the problem that this incurs, such as register allocation and various optimization techniques.

As our work focuses on the high-level aspects of gaining an RTCG system, the issues we address are largely orthogonal to recent work in the RTCG field. Instead, they are a direct continuation of previous work in the partial evaluation community. In some sense we are aiming to provide the missing link to systems like FABIUS. As such, our implementation is geared more towards applications well-known in partial evalution which benefit from RTCG rather than the other way around. Clearly, work remains to apply our approach to realistic RTCG systems which generate native code. The applicability of partial evaluation methodology as a framework for on-the-fly code generation has already been demonstrated [8, 9, 39].

Hence, the following issues need to be addressed:

- Our method needs to be applied to a compiler generating native code. A first step in our present system would be circumventing the intermediate representation which would speed up code generation significantly.
- Typical current RTCG applications must be reformulated in the context of incremental specialization [58]. The performance of those systems needs to be compared to hand-written ones.
- As mentioned in Sec. 7, the major obstacle to replacing the a stock compiler by an RTCG system is that the code generators still have to be loaded–and thus compiled by the stock compiler—before they can be executed. The obvious way to speed up that process is to apply the system to itself, and generate the generating extensions as object code themselves.

10 Conclusion

We have demonstrated that modern program transformation techniques like deforestation and partial evaluation are powerful tools in the hands of the programmer. We have successfully composed a partial evaluator with a compiler without rewriting everything from scratch. Instead we have reused code from the compiler and from the specializer and have generated the glue code automatically (by partial evaluation) from an annotated version of the original compiler. Our work is an attempt to bridge the gap between partial evaluation and run-time code generation methodologies.

Acknowledgments We would like to thank Richard Kelsey for explaining the internals of the Scheme 48 system.

References

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

[2] Anders Bondorf. Improving binding times without explicit CPS-conversion. In LFP 1992 [40], pages 1–10.

[3] Anders Bondorf. *Similix 5.0 Manual*. DIKU, University of Copenhagen, May 1993.

[4] Wei-Ngan Chin. Safe fusion of functional expressions. In LFP 1992 [40], pages 11–20.

[5] Wei-Ngan Chin. Safe fusion of functional expressions II: Further improvements. *Journal of Functional Programming*, 4(4):515–555, October 1994.

[6] Charles Consel and Olivier Danvy. For a better support of static data flow. In Hughes [27], pages 496–519.

[7] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proc. 20th Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

[8] Charles Consel, Luke Hornof, François Noël, Jacques Noyé, and Nicolae Volanschi. A uniform approach for compile-time and run-time specialization. In Danvy et al. [12], pages 54–72.

[9] Charles Consel and Francois Noël. A general approach for run-time specialization and its application to C. In POPL1996 [48], pages 145–156.

[10] Olivier Danvy. Pragmatics of type-directed partial evaluation. In Danvy et al. [12], pages 73–94.

[11] Olivier Danvy. Type-directed partial evaluation. In POPL1996 [48], pages 242–257.

[12] Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Dagstuhl Seminar on Partial Evaluation 1996*, number 1110 in Lecture Notes in Computer Science, Schloß Dagstuhl, Germany, February 1996. Springer-Verlag.

[13] Olivier Danvy and Rene Vestergaard. Semantics-based compiling: A case study in type-directed partial evaluation. Technical Report BRICS-RS-96-13, BRICS, May 1996.

[14] Scott Draves. Compiler generation for interactive graphics using intermediate code. In Danvy et al. [12], pages 95–114.

[15] Dirk Dussart and Peter Thiemann. Partial evaluation for higher-order languages with state. Berichte des Wilhelm-Schickard-Instituts WSI-96-XX, Universität Tübingen, November 1996.

[16] R. Kent Dybvig, editor. *International Conference on Functional Programming*, Philadelphia, PA, May 1996. ACM Press, New York.

[17] Dawson R. Engler and Todd A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 263–272. ACM Press, 1994.

[18] David Espinosa. *Semantic Lego*. PhD thesis, Columbia University, 1995.

[19] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proc. of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 237–247, Albuquerque, New Mexico, June 1993.

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

[21] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In Arvind, editor, *Proc. Functional Programming Languages and Computer Architecture 1993*, pages 223–232, Copenhagen, Denmark, June 1993. ACM Press, New York.

[22] Robert Glück and Jesper Jørgensen. Constraint-based multi-level binding-time analysis of higher-order languages. working note.

[23] Geoffrey W. Hamilton. Higher order deforestation. In Kuchen and Swierstra [34], pages 213–227.

[24] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In *Proc. 21st Annual ACM Symposium on Principles of Programming Languages*, pages 458–471, Portland, OG, January 1994. ACM Press.

[25] Urs Hölzle and David Ungar. Opimizing dynamically-dispatched calls with run-time type feedback. In *Proc. of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 326–335, Orlando, Fla, USA, June 1994. ACM Press.

[26] Carsten Kehler Holst. Language triplets: The AMIX approach. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 167–186, Amsterdam, 1988. North-Holland.

[27] John Hughes, editor. *Functional Programming Languages and Computer Architecture*, number 523 in Lecture Notes in Computer Science, Cambridge, MA, 1991. Springer-Verlag.

[28] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proc. Functional Programming Languages and Computer Architecture 1985*, number 201 in Lecture Notes in Computer Science. Springer-Verlag, 1985.

[29] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[30] Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. In *Proc. 16th Annual ACM Symposium on Principles of Programming Languages*, pages 281–292, Austin, Texas, January 1989. ACM Press.

[31] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1995.

[32] David Keppel, Susan J. Eggers, and Robert R. Henry. A case for runtime code generation. Technical Report 91-11-04, University of Washington, Seattle, WA, 1991.

[33] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for Scheme. *SIGPLAN Notices*, 21(7):219–233, July 1986. Proc. Sigplan '86 Symp. on Compiler Construction.

[34] Herbert Kuchen and Doaitse Swierstra, editors. *International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '96)*, number 1140 in Lecture Notes in Computer Science, Aachen, Germany, September 1996. Springer-Verlag.

[35] John Launchbury and Carsten Kehler 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.

[36] John Launchbury and Tim Sheard. Warm fusion: Deriving build-cata's from recursive definitions. In Peyton Jones [46], pages 314–323.

[37] Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. In *Proc. 1994 ACM Conference on Lisp and Functional Programming*, pages 227–238, Orlando, Florida, USA, June 1994. ACM Press.

[38] Peter Lee and Mark Leone. Optimizing ML with run-time code generation. In *Proc. of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 137–148, Philadelphia, PA, USA, May 1996. ACM Press.

[39] Mark Leone and Peter Lee. Lightweight run-time code generation. In Peter Sestoft and Harald Søndergaard, editors, *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM '94*, pages 97–106, Orlando, Fla., June 1994. University of Melbourne, Australia. Technical Report 94/9, Department of Computer Science.

[40] *Proc. 1992 ACM Conference on Lisp and Functional Programming*, San Francisco, California, USA, June 1992.

[41] Grant Malcolm. Homomorphisms and promotability. In J. L. A. van de Snepscheut, editor, *Mathematics of Program Construction*, pages 335–347, 1989. LNCS 375.

[42] Henry Massalin. *Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, 1992.

[43] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In Hughes [27], pages 124–144.

[44] Kristian Nielsen and Morten Heine Sørensen. Call-by-name CPS-translation as a binding-time improvement. In Alan Mycroft, editor, *Proc. International Static Analysis Symposium, SAS'95*, number 983 in Lecture Notes in Computer Science, pages 296–313, Glasgow, Scotland, September 1995. Springer-Verlag.

[45] Kristian Nielsen and Morten Heine Sørensen. Deforestation, partial evaluation, and evaluation orders. In Doaitse Swierstra and Manuel Hermenegildo, editors, *International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '95)*, number 982 in Lecture Notes in Computer Science, Utrecht, The Netherlands, September 1995. Springer-Verlag.

[46] Simon Peyton Jones, editor. *Functional Programming Languages and Computer Architecture*, La Jolla, CA, June 1995. ACM Press, New York.

[47] Rob Pike, Bart Locanthi, and John Reiser. Trade-offs for bitmap graphics on the Blit. *Software—Practice & Experience*, 15:131–151, 1985.

[48] *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg, Fla., January 1996. ACM Press.

[49] Revised^4 report on the algorithmic language Scheme. *Lisp Pointers*, IV(3):1–55, July–September 1991.

[50] Amr Sabry and Philip Wadler. A reflection on call-by-value. In Dybvig [16], pages 13–24.

[51] David A. Schmidt. *Denotational Semantics, A Methodology for Software Development*. Allyn and Bacon, Inc, Massachusetts, 1986.

[52] Helmut Seidl. Integer constraints to stop deforestation. In Hanne Riis Nielson, editor, *Proc. 6th European Symposium on Programming*, number 1058 in Lecture Notes in Computer Science, Linköping, Sweden, April 1996. Springer-Verlag.

[53] Morten Heine Sørensen. A grammar-based data-flow analysis to stop deforestation. In Sophie Tison, editor, *Proc. Trees in Algebra and Programming*, number 787 in Lecture Notes in Computer Science, pages 335–351, Edinburgh, UK, April 1994. Springer-Verlag.

[54] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In Donald Sannella, editor, *Proc. 5th European Symposium on Programming*, number 788 in Lecture Notes in Computer Science, pages 485–500, Edinburgh, UK, April 1994. Springer-Verlag.

[55] Guy L. Steele. Rabbit: a compiler for Scheme. Technical Report AI-TR-474, MIT, Cambridge, MA, 1978.

[56] Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In Peyton Jones [46], pages 306–313.

[57] Peter Thiemann. Cogen in six lines. In Dybvig [16], pages 180–189.

[58] Peter Thiemann. Implementing memoization for partial evaluation. In Kuchen and Swierstra [34], pages 198–212.

[59] Peter Thiemann and Dirk Dussart. Partial evaluation for higher-order languages with state. Berichte des Wilhelm-Schickard-Instituts WSI-97-XX, Universität Tübingen, April 1997.

[60] Mads Tofte. *Compiler Generators—What They Can Do, What They Might Do and What They Probably Never Do*, volume 19. Springer-Verlag, March 1990.

[61] Valentin F. Turchin. A supercompiler system based on the language Refal. *SIGPLAN Notices*, 14(2):46–54, February 1979.

[62] Philip L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.