Realistic Compilation By Partial Evaluation

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

Abstract

Two key steps in the compilation of strict functional languages are the conversion of higher-order functions to data structures (*closures*) and the transformation to tail-recursive style. We show how to perform both steps at once by applying first-order offline partial evaluation to a suitable interpreter. The resulting code is easy to transliterate to low-level C or native code. We have implemented the compilation to C; it yields a performance comparable to that of other modern Scheme-to-C compilers. In addition, we have integrated various optimizations such as constant propagation, higher-order removal, and arity raising simply by modifying the underlying interpreter. Purely first-order methods suffice to achieve the transformations. Our approach is an instance of semantics-directed compiler generation.

Keywords semantics-directed compiler generation, partial evaluation, compilation of higher-order functional languages

Partial evaluation is an automatic program transformation that performs aggressive constant propagation [28, 18]. When applied to an interpreter with respect to a constant ("static") input program for the interpreter, partial evaluation performs compilation into the target language of the partial evaluator. Naive interpreters subjected to offline partial evaluation usually produce straightforward compiled code. Moreover, if the input language of the interpreter, and the input and output languages of the partial evaluator are identical—that is, if it is a self-interpreter—the compilation is essentially the identity function.

However, if the interpreter uses only a subset of the subject language, so do the compiled programs. In addition, after changing the interpreter to propagate more information statically, the produced compiler performs optimization. The generation of optimizing program transformers by partial evaluation is called the *interpretive approach* [24]. It has been applied to a wide range of problems: to the generation of optimizing specializers, supercompilers, and deforesters [24, 25]—albeit in the context of first-order languages.

We show that the interpretive approach can achieve optimizing compilation of a strict, higher-order functional language. Our compilation system consists of two parts: an optimizing transformer, which translates higher-order recursion equations into first-order tail-recursive Scheme programs, generated automatically from a suitable interpreter by partial evaluation, and a simple, hand-written translator from first-order tail-recursive Scheme into low-level C. It is also possible to generate the back-end translator by the same method from an interpreter, using a partial evaluator for C [2]. Hence, partial evaluation offers the development of a Scheme compiler for the price of writing two interpreters. The automatic conversion to tail form is also the first solution to Jones's 1987 challenge 11.5 [26].

The optimizing compiler performs aggressive constant propagation and higher-order removal; it is a specializer in its own right. For its generation, we exploit two principles: the *specializer projections* for the generation of the transformer, and the *language preservation property* of offline partial evaluators for the translation of higher-order programs into first-order tail-recursive code.

We have generated the transformer from an interpreter using the partial evaluator Unmix. Unmix, a descendant of the Moscow specializer [36], dating back to 1990, treats only a first-order subset of Scheme, and does not handle partially static data structures. Since our transformer performs a much more powerful specialization on higher-order Scheme, and does handle partially static data structures, we have achieved a bootstrapping effect.

Our work also refutes the 1991 claim of Consel and Danvy [17] that realistic compiler generation by partial evaluation is only possible through recent advances in partial evaluation technology. We show that neither higher-order specialization nor partially static data structures are vital to achieve realistic compilation. A simple first-order partial evaluator suffices to do the job, even for a higher-order subject language.

Overview We start with a small example for specialization and translation into tail-recursive code in Sec. 1. Section 2 is a brief introduction to partial evaluation. In Sec. 3 we explain the two basic principles needed to generate stand-alone compilers by partial evaluation: the *specializer projections* and the *language preservation property*. Section 4 shows how to turn a simple-minded recursive-descent interpreter into a two-level interpreter from which the partial evaluator produces the optimizing compiler. In Sec. 5, we describe our approach to generating C code from the compiler output by hand, followed by a recipe on how to achieve the same effect by using partial evaluation again. Section 6 presents experimental results, and Sec. 7 discusses related work.

1 Examples

We illustrate the transformations that our compiler performs by applying them to a version of append written in continuation-passing style (CPS):

```
(define (append x y)
```

Appears in: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'96, Philadelphia, USA.

```
 (cps-append x y (lambda (x) x)))

(define (cps-append x y c)
 (if (null? x)
    (c y)
    (cps-append (cdr x) y
            (lambda (xy)
              (c (cons (car x) xy))))))
```

The compiler converts the program to first-order tail-recursive Scheme. It residualizes the `lambda` appearing in the program, and represents the resulting functions by closures. Closures consist of a closure label identifying its originating expression, and the values of their free variables. They are constructed by `make-closure` and accessed by `closure-label` and `closure-freeval` The closure label `10` denotes the identity, `24` the inner continuation. Whenever the program applies a closure, it dispatches on the `closure-label` component. This happens for both applications of the continuation `c`, once in `s1-eval-$3` for `(c y)` and once in `s1-eval-$9` for the other application. Note that the identifier names in the residual program have been replaced by generic names from the interpreter. Namely, the counterparts to the original identifiers now have the form `cv-vals-`*xx*.

```
(define (append x y)
 (s1-eval-$3 (make-closure 10) y x))

(define (s1-eval-$3 cv-vals-$1 cv-vals-$2 cv-vals-$3)
 (if (null? cv-vals-$3)
  (if (equal? 10 (closure-label cv-vals-$1))
    cv-vals-$2
    (do-closure-cv-bindings-$2 cv-vals-$2 cv-vals-$1))
  (s1-eval-$3
    (make-closure 24 cv-vals-$1 cv-vals-$3)
    cv-vals-$2
    (cdr cv-vals-$3))))

(define (do-closure-cv-bindings-$2 first-val closure)
 (s1-eval-$9
  (closure-freeval closure 1)
  first-val
  (closure-freeval closure 0)))

(define (s1-eval-$9 cv-vals-$1 cv-vals-$2 cv-vals-$3)
 (if (equal? 10 (closure-label cv-vals-$3))
  (cons (car cv-vals-$1) cv-vals-$2)
  (do-closure-cv-bindings-$2
    (cons (car cv-vals-$1) cv-vals-$2)
    cv-vals-$3)))
```

When given a known first argument `(foo bar)`, the compiler performs specialization:

```
(define (append-$1 y)
 (cons 'foo (cons 'bar y)))
```

The next step in the compilation is the translation to C. We have omitted actual output. Section 5 describes our C back end.

2 Partial Evaluation Issues

Partial evaluation is a specialization technique: If parts of the input of a *subject program* are known at compile time, a partial evaluator generates a *residual program* specialized with respect to the static input. The residual program only takes the remaining, *dynamic* parts of the input as parameters, and produces the same results as the subject program applied to the full input. Partial evaluation can remove interpretive overhead and produce significant speed-ups [28].

An *offline* partial evaluator consists of a *binding-time analysis* and a *reducer*. The binding-time analysis, applied to the subject program and the binding times of its arguments, annotates each expression in the program with a binding time, static or dynamic. The reducer processes the annotated program and the static part of the input, reducing static expressions and rebuilding dynamic ones, driven by the annotations. Whereas simple-minded binding-time analyses only handle the binding times "completely static" and "completely dynamic," more sophisticated variants also treat *partially static data* [33, 32, 9, 16].

In contrast, *online* partial evaluators [48, 38] are one-pass programs which decide "online" whether to reduce or rebuild an expression. They are generally more powerful than their offline counterparts because they exploit information about actual values—rather than only their binding times—to decide whether to reduce or rebuild.

For our experiments, we use Unmix, a simple offline partial evaluator for a first-order, purely functional subset of Scheme. Unmix employs classic Mix technology [29], and does not handle partially static data. However, its post-processor performs arity raising [37] which is crucial to the generation of efficient residual programs in the absence of partially static data.

3 Prerequisites for Compiler Generation

The interpreters described here exploit two basic principles: The *specializer projections* specify how to generate specializers from interpreters, and the *language-preservation property* of offline partial evaluation is the basis for higher-order removal and conversion to tail form.

The Specializer Projections Partial evaluation of interpreters can perform compilation. The specification of an S-interpreter `int` written in L is

$$[\![\texttt{int}]\!]_L\ P_S\ inp = [\![P_S]\!]_S\ inp$$

where $[\![\_]\!]_L$ is the execution of an L-program, P_S is an S-program, and inp is its input. An $L{\rightarrow}L$-partial evaluator `pe` can compile P_S into an equivalent L-program P_L such that $[\![P_L]\!]_L\, inp = [\![P_S]\!]_S\, inp$ as described by the first *Futamura projection* [21]:

$$P_L = [\![\texttt{pe}]\!]_L\ \texttt{int}^{sd}\ P_S.$$

The sd superscript to `int` means that the partial evaluator is to treat the first argument of `int` as static, the second as dynamic.

Exploiting repeated self-application, the second and third Futamura projections describe the generation of compilers and compiler generators [28].

A generalization of the Futamura projections shows how to generate specializers, or constant-propagating optimizers from a *two-level interpreter* [24, 25] `2int` which accepts the input to the interpreted program in two parts: one static and one dynamic. The interpreter tries to perform each operation with the static part of the input first; only if that fails, the dynamic part is consulted. Residual programs result from the first *specializer projection* [23, 25]:

$$R_L = [\![\texttt{pe}]\!]\ \texttt{2int}^{ssd}\ P_S\ inp_s$$

where inp_s is the static part of the input and R_L is the specialized program. Analogous to the Futamura projections, stand-alone specializers and specializer generators result result from the second and third specializer projection.

We will introduce an ordinary one-level interpreter and then show how to extend it to two levels.

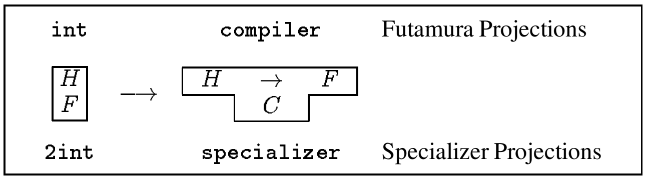

Figure 1: Generation of compilers and specializers

The Language Preservation Property Mix-style offline partial evaluators have the *language preservation property* which is obvious from inspecting their specialization phase [28].

> For any sublanguage $L' \subseteq L$ which includes all L'-computable values as literals, and for any binding-time-annotated L-program P every dynamic expression of which belongs to L', $[\![\texttt{pe}]\!]\ P\ x \in L'$ holds for arbitrary static x.

Specialization of an interpreter can translate higher-order to first-order programs: Suppose `pe` is a partial evaluator for a subset C of Scheme. The first-order language F in which the interpreter is written has $F \subseteq C$. Finally, the interpreter itself executes programs in the higher-order Scheme subset H. See Fig. 1 for an illustration. Because `pe` preserves the F-ness of the subject program, the residual programs

$$P_F = [\![\texttt{pe}]\!]\ \texttt{int}^{sd}\ P_H = [\![\texttt{compiler}]\!] P_H$$

—the compiled program—and

$$R_F = [\![\texttt{pe}]\!]\ \texttt{2int}^{ssd}\ P_H\ inp = [\![\texttt{specializer}]\!] P_H\, inp$$

—the specialized program—are F-programs.

4 Deriving the Interpreter

We start from a straightforward, environment-based interpreter and transform it step by step:

- By subjecting the interpreter to closure conversion, the generated transformer performs closure conversion as well.
- Converting the interpreter to tail form leads to a transformer into tail form.
- Next, adding constant propagation in static data turns the transformer into an optimizer.
- Finally, we introduce a generalization strategy to ensure termination.

4.1 A Straightforward Interpreter

Figure 2 defines the syntax of the purely functional Scheme subset treated by our interpreters. For the sake of simplicity, we have restricted it to lambda abstractions of one argument.

Figure 3 shows a standard interpreter for the Scheme subset. The meta-language is a call-by-value lambda calculus enriched with constants, sums, and products. $T \rightarrow E_1 \mid E_2$ denotes the McCarthy conditional. The notation $\mathsf{Value}^* \rightarrow \mathsf{Value}$ is a shorthand for the sum of $() \rightarrow \mathsf{Value}$, $\mathsf{Value} \rightarrow \mathsf{Value}$, $\mathsf{Value} \times \mathsf{Value} \rightarrow \mathsf{Value}$ etc. We have omitted the injections and case analysis for elements of Value. We assume that each expression is uniquely labeled by an $\ell \in \mathsf{Label}$. Where necessary, we indicate the label by a superscript. ψ serves for both function and label lookup.

$$\begin{array}{rcl}
 & V \in & \mathsf{Variable} \\
 & P \in & \mathsf{ProcName} \\
 & O \in & \mathsf{Operators} \\
 & K \in & \mathsf{Constants} \\
 & E \in & \mathsf{Expression} \\
 & D \in & \mathsf{Definition} \\
 & \Pi \in & \mathsf{Program}
\end{array}$$

$$\begin{array}{rcl}
E & ::= & V \mid K \mid (\mathsf{if}\ E\ E\ E) \mid (O\ E^*) \mid (P\ E^*) \mid \\
 & & (\mathsf{let}\ ((V\ E))\ E) \mid (\mathsf{lambda}\ (V)\ E) \mid (E\ E) \\
D & ::= & (\mathsf{define}\ (P\ V^*)\ E) \\
\Pi & ::= & D^+
\end{array}$$

Figure 2: Syntax

$$\begin{array}{lcl}
\ell \in \mathsf{Label} & & \\
\mathsf{Value} & = & \mathsf{BaseValue} + (\mathsf{Value} \rightarrow \mathsf{Value}) \\
\rho \in \mathsf{Env} & = & \mathsf{Variable} \rightarrow \mathsf{Value} \\
\psi \in \mathsf{ProcEnv} & = & (\mathsf{ProcName} + \mathsf{Label}) \rightarrow \mathsf{Expression} \\
\mathcal{K}[\![_]\!] & : & \mathsf{Constants} \rightarrow \mathsf{Value} \\
\mathcal{O}[\![_]\!] & : & \mathsf{Operators} \rightarrow \mathsf{Value}^* \rightarrow \mathsf{Value} \\
\mathcal{E}[\![_]\!] & : & \mathsf{Expression} \rightarrow \mathsf{Env} \rightarrow \mathsf{Value}
\end{array}$$

$$\begin{array}{ll}
\mathcal{E}[\![V]\!]\rho & = \rho[\![V]\!] \\
\mathcal{E}[\![K]\!]\rho & = \mathcal{K}[\![K]\!] \\
\mathcal{E}[\![(\mathsf{if}\ E_1\ E_2\ E_3)]\!]\rho & = \mathcal{E}[\![E_1]\!]\rho \rightarrow \mathcal{E}[\![E_2]\!]\rho \mid \mathcal{E}[\![E_3]\!]\rho \\
\mathcal{E}[\![(O\ E_1 \ldots E_n)]\!]\rho & = \mathcal{O}[\![\mathrm{O}]\!](\mathcal{E}[\![E_1]\!]\rho, \ldots, \mathcal{E}[\![E_n]\!]\rho) \\
\mathcal{E}[\![(P\ E_1 \ldots E_n)]\!]\rho & = \mathcal{E}[\![\psi(\mathrm{P})]\!][V_i \mapsto \mathcal{E}[\![E_i]\!]\rho] \\
\mathcal{E}[\![(\mathsf{let}\ ((V\ E_1))\ E_2)]\!]\rho & = \mathcal{E}[\![E_2]\!]\rho[V \mapsto \mathcal{E}[\![E_1]\!]\rho] \\
\mathcal{E}[\![(\mathsf{lambda}\ (V)\ E)]\!]\rho & = \lambda y.\mathcal{E}[\![E]\!]\rho[V \mapsto y] \\
\mathcal{E}[\![(E_1\ E_2)]\!]\rho & = (\mathcal{E}[\![E_1]\!]\rho)(\mathcal{E}[\![E_2]\!]\rho)
\end{array}$$

Figure 3: A standard call-by-value interpreter

4.2 Removing λ

As the first step towards true compilation, we apply Reynolds's defunctionalization [35], and change the representation of functions in the interpreter to *closures* consisting of the label of the originating lambda expression, and the values of its free variables (see Fig. 4). (*freevars*(ℓ) computes the list of the free variables of the expression at ℓ in an arbitrary but fixed order.) Consequently, we now have a first-order interpreter for a higher-order language.

$$\begin{array}{rcl}
\mathsf{Closure} & = & \mathsf{Label} \times \mathsf{Value}^* \\
\mathsf{Value} & = & \mathsf{BaseValue} + \mathsf{Closure}
\end{array}$$

$$\begin{array}{ll}
\mathcal{E}[\![(\mathsf{lambda}^{\ell}\ (V)\ E)]\!]\rho = & \mathit{let}\ V_1 \ldots V_n = \mathit{freevars}(\ell) \\
 & \quad \mathit{in}\ (\ell, \rho V_1 \ldots \rho V_n) \\
\mathcal{E}[\![(E_1\ E_2)]\!]\rho = & \mathit{let}\quad (\ell, v_1 \ldots v_n) = \mathcal{E}[\![E_1]\!]\rho \\
 & \quad\quad (\mathsf{lambda}\ (V)\ E) = \psi(\ell) \\
 & \quad\quad V_1 \ldots V_n = \mathit{freevars}(\ell) \\
 & \quad \mathit{in}\ \mathcal{E}[\![E]\!][V \mapsto \mathcal{E}[\![E_2]\!]\rho, \\
 & \quad\quad V_1 \mapsto v_1, \ldots V_n \mapsto v_n]
\end{array}$$

Figure 4: Changes to interpreter after closure conversion

The interpreter shown in Fig. 4 does not specialize effectively yet. On closure application, the label ℓ is dynamic. Hence, the lambda expression in the program text would normally be dynamic as well which would lead to unwanted interpretive overhead in the specialized code. Instead, the actual interpreter employs a *binding-time improvement* to make the expression argument static again—called "The Trick" [28]: On closure application, the interpreter loops over all lambda expressions that could have generated the closure to be applied, comparing each one with ℓ successively. When the interpreter finds the lambda belonging to ℓ, it continues interpretation with the now static expression. The interpreter employs a simple equational flow analysis [11] to restrict the set of lambdas which it needs to test. The residual code then performs a sequential dispatch on closure application.

4.3 Converting to Tail Form

In the next step we convert the interpreter to tail-recursive style. Again, by changing the interpreter, the generated compiler performs the corresponding transformation. In a higher-order setting, we would transform the interpreter into CPS [35, 20, 4]. CPS makes control flow explicit by encoding the current evaluation context as a function. As we only have first-order methods at our disposal, we encode evaluation contexts in the same way as functions: by closures. Thus, we encode the current evaluation context directly as a function, avoiding an explicit CPS transformation.

$$\begin{array}{lcl} E & ::= & SE \mid (\mathsf{if}\ SE\ E\ E) \mid (P\ SE^*) \mid (SE\ E) \\ SE & ::= & V \mid K \mid (O\ SE^*) \mid (\mathsf{lambda}\ (V)\ E) \end{array}$$

Figure 5: Desugared syntax

In our interpreter, a desugaring phase reduces the number of different evaluation contexts to one—the application of a closure. In non-tail positions we only allow *simple expressions* which evaluate directly to values—constants, variables, applications of primitives, and lambda abstractions. Figure 5 shows the simplified syntax. In the specification, SE is for simple expressions. The desugaring phase simply moves the non-tail expressions into parameters to lambda abstractions. Thus, the expression $(f(g\ x))$ becomes $((\mathsf{lambda}(r)(f\ r))(g\ x))$. In addition, the desugarer replaces lets by equivalent applications of lambda abstractions.

The tail-recursive interpreter is shown in Fig. 6. $\mathcal{S}$ evaluates simple expressions. $\mathcal{E}^*$ evaluates "serious" expressions. $\mathcal{E}^*$ has an additional argument, a context stack, which keeps track of pending context closures. When $\mathcal{E}^*$ reaches a simple expression SE, it evaluates SE via $\mathcal{S}$, and passes the result to $\mathcal{C}$ which processes the stack of pending contexts. $\mathcal{C}$ applies the closure on top. If the context stack is empty, $\mathcal{C}$'s argument is the final result of the interpretation.

$\mathcal{S}$ need not be tail-recursive: All calls to $\mathcal{S}$ are statically unfoldable, and consequently never perform function calls. Hence, partially evaluating the interpreter in Fig. 6 yields tail-recursive residual programs.

4.4 Propagating Constants

Now we turn the transformer into an optimizer to first-order tail-recursive code. We split the environment ρ into a static and a dynamic part, converting the interpreter into a two-level interpreter and making it amenable to the specializer projections. To support partially static data structures, we change ρ to associate names with completely static *value descriptions* instead of dynamic values. A value description may represent an arbitrary partially static data object:

$$desc ::= \mathtt{quote}(K) \mid \mathtt{cons}(desc, desc) \mid \mathtt{clos}(\ell, desc^*) \mid \mathtt{cv}(i)$$

A value description can be a completely static atomic value (`quote`), a pair of value descriptions (`cons`), a partially static closure (`clos`), or a *configuration variable* [24, 47, 45] whose value is stored in a separate environment σ. $fresh(\sigma)$ yields an unused configuration variable.

Figure 7 shows the two-level interpreter with the following functions:

$\mathcal{S}^*[\![\_]\!]$ evaluates a simple expression to a value description. The constructors cons and lambda evaluate to the corresponding descriptions. For selector and primitive applications, the interpreter first examines if it can reduce them statically—for example, when car is applied to a `cons` description. If that is not possible, the interpreter generates a new configuration variable and maps it to the dynamic result of the expression. Therefore, $\mathcal{S}^*[\![\_]\!]$ returns a new configuation variable environment along with the value description. Again, all non-tail calls in the definition are statically unfoldable.

Note that a simple expression is static if all its free variables refer to static value descriptions (those that do not contain `cv` components). For static simple expressions, $\mathcal{S}^*[\![\_]\!]$ always produces a static value.

$\mathcal{D}[\![\_]\!]$ evaluates an arbitrary value description to a value.

$\mathcal{E}^\diamond[\![\_]\!]$ is the main evaluation function. It is analogous to the $\mathcal{E}^*[\![\_]\!]$ function in the simple tail-recursive interpreter in Fig. 6. The main difference is in the handling of if: The interpreter tries to determine the conditional statically first. Only if that fails, it introduces a residual conditional. Our implementation can actually infer a static if more often than the interpreter shown in Fig. 6, for example on null? tests on `cons` descriptions with dynamic components.

$\mathcal{C}^\diamond$ handles context application, analogous to $\mathcal{C}$ in Fig. 6. $\mathcal{C}^\diamond$ also needs to distinguish between static and dynamic contexts. For static contexts, it is trivial to prepare a suitable environment and continue evaluation. For dynamic contexts, the interpreter needs to introduce new configuration variables for their (dynamic) free variables.

The interpreter presented here is not yet suitable for successful offline partial evaluation. Some standard binding-time improvements [28] are necessary to ensure that ρ and γ as well as the expression to be evaluated stay static. For instance, the interpreter also performs "The Trick" on the application of a dynamic context, just as the interpreter shown in Fig. 4.

4.5 Addressing Non-Termination Woes

The two-level interpreter exhibited in the last section is first-order, tail-recursive, and performs constant propagation. However, partial evaluation with respect to a static input program does not terminate for non-tail-recursive input programs: Mix-style partial evaluators such as Unmix do not detect and properly handle static data structures that grow without bounds under dynamic control. Our interpreter propagates such data in three places:

1. The stack of evaluation contexts may contain a context that leads to its own repeated evaluation.
2. A closure may contain a closure generated from the same lambda expression as part of the value of one of its free variables.
3. Applications of cons may nest.

Exactly these conditions lead to self-embedding data structures which potentially grow infinitely. The critical data structures must be *generalized* (coerced to dynamic values) which removes their static value from the view of the partial evaluator. For closures and data structures, generalization is straightforward: The interpreter replaces the offending value descriptions by fresh `cv` descriptions, and adds the generalized values to σ. To handle dynamic evaluation contexts, we must split the context stack into a static part and a

$$
\begin{array}{lcl}
\gamma \in \ \mathsf{Context} & = & \mathsf{Closure}^* \\
\mathcal{S}[\![\_]\!] & : & \mathsf{SimpleExpression} \to \mathsf{Env} \to \mathsf{Value} \\
\mathcal{E}^*[\![\_]\!] & : & \mathsf{Expression} \to \mathsf{Env} \to \mathsf{Context} \to \mathsf{Value} \\
\mathcal{C} & : & \mathsf{Value} \to \mathsf{Context} \to \mathsf{Value} \\
\mathcal{S}[\![V]\!]\rho & = & \rho V \\
\mathcal{S}[\![K]\!]\rho & = & \mathcal{K}[\![K]\!] \\
\mathcal{S}[\![(O\ SE_1\ \ldots\ SE_n)]\!]\rho & = & \mathcal{O}[\![O]\!](\mathcal{S}[\![SE_1]\!]\rho, \ldots, \mathcal{S}[\![SE_n]\!]\rho) \\
\mathcal{S}[\![(\mathsf{lambda}^\ell\ (V)\ E)]\!]\rho & = & \mathit{let}\ V_1 \ldots V_n = \mathit{freevars}(\ell)\ \mathit{in}\ (\ell, \rho V_1 \ldots \rho V_n) \\
\\
\mathcal{E}^*[\![SE]\!]\rho\gamma & = & \mathcal{C}(\mathcal{S}[\![SE]\!]\rho)\gamma \\
\mathcal{E}^*[\![(\mathsf{if}\ SE\ E_1\ E_2)]\!]\rho\gamma & = & \mathcal{S}[\![SE]\!]\rho \to \mathcal{E}^*[\![E_1]\!]\rho\gamma \mid \mathcal{E}^*[\![E_2]\!]\rho\gamma \\
\mathcal{E}^*[\![(P\ SE_1\ \ldots\ SE_n)]\!]\rho\gamma & = & \mathcal{E}^*[\![\psi(P)]\!][V_1 \mapsto \mathcal{S}[\![SE_1]\!]\rho, \ldots, V_n \mapsto \mathcal{S}[\![SE_n]\!]\rho]\gamma \\
\mathcal{E}^*[\![(SE\ E)]\!]\rho\gamma & = & \mathcal{E}^*[\![E]\!]\rho(\mathcal{S}[\![SE]\!]\rho : \gamma) \\
\\
\mathcal{C}v((\ell, v_1 \ldots v_n) : \gamma') & = & \mathit{let}\ (\mathsf{lambda}\ (V)\ E) = \psi(\ell) \\
 & & \qquad V_1 \ldots V_n = \mathit{freevars}(\ell) \\
 & & \mathit{in}\ \mathcal{E}^*[\![E]\!][V \mapsto v, V_1 \mapsto v_1, \ldots, V_n \mapsto v_n]\gamma' \\
\mathcal{C}v[\,] & = & v
\end{array}
$$

Figure 6: Tail-recursive interpreter

dynamic part, and use the dynamic stack for critical contexts that may cause non-termination.

We have implemented an online strategy and an offline analysis for generalization:

Online Generalization Self-embedding data can only grow without bounds inside of the branches of dynamic conditionals and through bodies of dynamic lambdas [10]. Under the online strategy [46], our interpreter delays generalization until it encounters a dynamic conditional. In that case, the interpreter scans ρ and γ for critical data structures and closures, and generalizes them as described above. Evaluation continues using dynamic evaluation contexts.

Offline Generalization Analysis An alternative approach uses a flow analysis [40, 8] to determine statically which lambdas and which cons expressions may lead to critical data in the interpreter. The corresponding descriptions are generalized on creation. As for critical evaluation contexts, they are merely closures already caught by the analysis.

The online strategy is less conservative since it generalizes only under dynamic conditionals. It necessarily generalizes less and propagates more static information. However, the online approach delays the generalization too long: The interpreter can only detect self-embedding when it has already occurred. Consequently, the respective code is already part of the residual program. Thus, the underlying data structures and loops are unrolled at least once before generalization happens, leading to redundant code. This is a well-known problem in online partial evaluation [38].

5 Compilation to C

We describe two ways to achieve compilation to the C language. The first one describes a very simple translation implemented by hand. It has been implemented and used to obtain the experimental data presented below. The second one presents a more speculative approach which again employs partial evaluation to obtain a C program from our Scheme subset. It has not been carried out in practice.

5.1 By Hand

The output language S_0 of the partial evaluation process is a tail-recursive first-order subset of Scheme which has a simple transliteration to C. The translation of an S_0 program to C yields a single function `program`. Procedure headers are translated into labels, hence (tail-recursive) function calls turn out to be `goto`s.

Parameters are passed in a fixed number of variables local to `program`, but global to all procedures. On entry to a procedure, a new C scope is opened which declares the procedure's private parameter variables. Then the relevant global parameter variables are copied into the private variables.

Since procedure calls' arguments are simple expressions, there are no nested procedure calls in S_0. Therefore the arguments of a call can be evaluated without referring to the global parameter variables. Thus the construction of an argument list is straightforward: generate code to evaluate the simple argument expressions and assign the result to the respective global parameter variables. Finally, control is transferred to the next procedure by a `goto`.

The translation of simple expressions SE is an assignment of its value to a new temporary variable. Temporary variables are also local variables of `program`, but global to all procedures. Each temporary variable is defined and used exactly once. We rely on the C compiler's register allocator to merge variables (global parameter variables, procedure argument variables, and temporaries) if their life ranges are disjoint. The evaluation is sequentialized using C's sequential evaluation operator `(expr, expr)`. Thus the result of the translation is a C expression. All other expressions E are translated into C statements. For a simple expression a `return` statement is generated which terminates the execution of `program`.

The most important interface between the tail-recursive interpreter and the translation to C is the closure representation. The interpreter treats closures as an abstract datatype with operations `make-closure`, `closure-label`, and `closure-freeval` with the obvious interpretations. These operations are propagated to residual programs. The translator to C is free to choose an efficient implementation for closures. The current implementation uses a flat vector representation. Note that the C code also performs a sequential dispatch on closure applications exactly like the Scheme input programs. It might be desirable to perform closure application by an indirect `goto` statement as allowed by GCC [41]. However, since sequential dispatch is inherent in our approach, it would seem difficult to achieve this by straightforward means.

We represent Scheme data objects by a C `union`, and we employ the Boehm garbage collector for C [6]. There is no cooperation between the translation and the garbage collector.

$$
\begin{array}{rcl}
\rho \in \text{Env} & = & \text{Variable} \to \text{ValDesc} \\
\sigma \in \text{CVEnv} & = & \text{ConfigVariable} \to \text{Value} \\
\gamma \in \text{Context} & = & \text{ValDesc}^{*} \\
\mathcal{S}^{*}[\![\_]\!] & : & \text{SimpleExpression} \to \text{Env} \to \text{CVEnv} \to (\text{ValDesc} \times \text{CVEnv}) \\
\mathcal{D}[\![\_]\!] & : & \text{ValDesc} \to \text{CVEnv} \to \text{Value} \\
\mathcal{E}^{\diamond}[\![\_]\!] & : & \text{Expression} \to \text{CVEnv} \to \text{Context} \to \text{Value} \\
\mathcal{C}^{\diamond}\ \_ & : & \text{ValDesc} \to \text{CVEnv} \to \text{Context} \to \text{Value}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{S}^{*}[\![V]\!]\rho\sigma & = & (\rho V, \sigma) \\
\mathcal{S}^{*}[\![K]\!]\rho\sigma & = & (\mathtt{quote}(K), \sigma) \\
\mathcal{S}^{*}[\![(\text{cons } SE_1\ SE_2)]\!]\rho\sigma & = & \mathit{let}\ (desc_1, \sigma_1) = \mathcal{S}^{*}[\![SE_1]\!]\rho\sigma \\
 & & \quad (desc_2, \sigma_2) = \mathcal{S}^{*}[\![SE_2]\!]\rho\sigma_1 \\
 & & \mathit{in}\ (\mathtt{cons}(desc_1, desc_2), \sigma_2) \\
\mathcal{S}^{*}[\![(\text{car } SE)]\!]\rho\sigma & = & \mathit{let}\ (desc, \sigma') = \mathcal{S}^{*}[\![SE]\!]\rho\sigma \\
 & & \mathit{in}\ \ (desc = \mathtt{cons}(desc_1, desc_2)) \\
 & & \quad \to\ (desc_1, \sigma') \\
 & & \quad \mid\ \mathit{let}\ i = \mathit{fresh}(\sigma)\ \mathit{in}\ (\mathtt{cv}(i), \sigma[i \mapsto \mathcal{O}[\![\text{car}]\!](\mathcal{D}[\![desc]\!]\sigma')]) \\
\mathcal{S}^{*}[\![(\text{cdr } SE)]\!]\rho\sigma & = & \text{analogous} \\
\mathcal{S}^{*}[\![(\text{lambda}^{\ell}\ (V)\ \text{E})]\!]\rho\sigma & = & \mathit{let}\ V_1 \ldots V_n = \mathit{freevars}(\ell)\ \mathit{in}\ \mathtt{clos}(\ell, \rho V_1 \ldots \rho V_n) \\
\mathcal{S}^{*}[\![(O\ E_1\ \ldots\ E_n)]\!]\rho\sigma & = & \langle SE_1 \ldots SE_n \text{ static}\rangle \\
 & & \quad \to\ (\mathtt{quote}(\mathcal{O}[\![O]\!](\mathcal{D}[\![desc_1]\!]\sigma, \ldots, \mathcal{D}[\![desc_n]\!]\sigma)), \sigma) \\
 & & \quad \mid\ \mathit{let}\ i = \mathit{fresh}(\sigma) \\
 & & \qquad \mathit{in}\ (\mathtt{cv}(i), \sigma[i \mapsto \mathcal{O}[\![O]\!](\mathcal{D}[\![desc_1]\!]\sigma, \ldots, \mathcal{D}[\![desc_n]\!]\sigma)]) \\
\mathcal{D}[\![\mathtt{quote}(K)]\!]\sigma & = & \mathcal{K}[\![K]\!] \\
\mathcal{D}[\![\mathtt{cons}(desc_1, desc_2)]\!]\sigma & = & \mathcal{O}[\![\text{cons}]\!](\mathcal{D}[\![desc_1]\!]\sigma, \mathcal{D}[\![desc_2]\!]\sigma) \\
\mathcal{D}[\![\mathtt{clos}(\ell, desc_1 \ldots desc_n)]\!]\sigma & = & (\ell, \mathcal{D}[\![desc_1]\!]\sigma \ldots \mathcal{D}[\![desc_n]\!]\sigma) \\
\mathcal{D}[\![\mathtt{cv}(i)]\!]\sigma & = & \sigma(i) \\
 & & \\
\mathcal{E}^{\diamond}[\![SE]\!]\rho\sigma\gamma & = & \mathit{let}\ (desc, \sigma') = \mathcal{S}^{*}[\![SE]\!]\rho\sigma\ \mathit{in}\ \mathcal{C}^{\diamond}\ desc\ \sigma'\gamma \\
\mathcal{E}^{\diamond}[\![(\text{if } SE\ E_1\ E_2)]\!]\rho\sigma\gamma & = & \mathit{let}\ (desc, \sigma') = \mathcal{S}^{*}[\![SE]\!]\rho\sigma \\
 & & \mathit{in}\ \langle SE \text{ static}\rangle \\
 & & \quad \to\ (desc = \mathtt{quote}(\mathit{false})) \to \mathcal{E}^{\diamond}[\![E_2]\!]\rho\sigma\gamma \mid \mathcal{E}^{\diamond}[\![E_1]\!]\rho\sigma\gamma \\
 & & \quad \mid\ (\mathcal{D}[\![desc]\!]\sigma') \to \mathcal{E}^{\diamond}[\![E_1]\!]\rho\sigma\gamma \mid \mathcal{E}^{\diamond}[\![E_2]\!]\rho\sigma\gamma \\
\mathcal{E}^{\diamond}[\![(P\ SE_1\ \ldots\ SE_n)]\!]\rho\sigma\gamma & = & \mathit{let}\ (desc_1, \sigma_1) = \mathcal{S}^{*}[\![SE_1]\!]\rho\sigma \\
 & & \qquad \vdots \\
 & & \quad (desc_n, \sigma_n) = \mathcal{S}^{*}[\![SE_n]\!]\rho\sigma_{n-1} \\
 & & \mathit{in}\ \mathcal{E}^{\diamond}[\![\psi(P)]\!][V_1 \mapsto desc_1, \ldots, V_n \mapsto desc_n]\sigma_n\gamma \\
\mathcal{E}^{\diamond}[\![(SE\ E)]\!]\rho\sigma\gamma & = & \mathit{let}\ (desc, \sigma') = \mathcal{S}^{*}[\![SE]\!]\rho\sigma\gamma\ \mathit{in}\ \mathcal{E}^{\diamond}[\![E]\!]\rho\sigma'(desc : \gamma) \\
 & & \\
\mathcal{C}^{\diamond}\ desc\ \sigma(c : \gamma') & = & (c = \mathtt{clos}(\ell, desc_1 \ldots desc_n)) \\
 & & \quad \to\ \mathit{let}\ (\text{lambda}\ (V)\ E) = \psi(\ell) \\
 & & \qquad\qquad V_1 \ldots V_n = \mathit{freevars}(\ell) \\
 & & \qquad \mathit{in}\ \mathcal{E}^{\diamond}[\![E]\!][V \mapsto desc, V_1 \mapsto desc_1, \ldots, V_n \mapsto desc_n]\sigma\gamma' \\
 & & \quad \mid\ \mathit{let}\ (\ell, v_1 \ldots v_n) = \mathcal{D}[\![c]\!]\sigma \\
 & & \qquad\qquad (\text{lambda}\ (V)\ E) = \psi(\ell) \\
 & & \qquad\qquad V_1 \ldots V_n = \mathit{freevars}(\ell) \\
 & & \qquad\qquad i_1 = \mathit{fresh}(\sigma) \\
 & & \qquad\qquad desc_1 = \mathtt{cv}(i_1) \\
 & & \qquad\qquad \sigma_1 = \sigma[i_1 \mapsto v_1] \\
 & & \qquad\qquad\quad \vdots \\
 & & \qquad\qquad i_n = \mathit{fresh}(\sigma_{n-1}) \\
 & & \qquad\qquad desc_n = \mathtt{cv}(i_n) \\
 & & \qquad\qquad \sigma_n = \sigma_{n-1}[i_n \mapsto v_n] \\
 & & \qquad \mathit{in}\ \mathcal{E}^{\diamond}[\![E]\!][V \mapsto desc, V_1 \mapsto desc_1, \ldots, V_n \mapsto desc_n]\sigma_n\gamma' \\
\mathcal{C}^{\diamond}\ desc\ \sigma[\,] & = & \mathcal{D}[\![desc]\!]\sigma
\end{array}
$$

Figure 7: Two-level interpreter

5.2 By Partial Evaluation

The recent advent of C specializers [1, 2] facilitates a development which culminates in a complete compiler written entirely in C. As ingredients we only have to provide two interpreters, `int-s`, the higher-order to first-order interpreter which has been developed in Section 4, and `int-c`, a hypothetical interpreter for first-order tail-recursive Scheme written in C. Our tools are the compiler generator `cogen` derived by self-application from the Unmix specializer and the compiler generator `C-mix` [2]. Let $[\![\_]\!]_S$ denote execution of

Scheme programs and $[\![\_]\!]_C$ execution of C programs.

First, we apply

$$\texttt{gen-s} = [\![\texttt{cogen}]\!]_S \ \texttt{int-s}$$

to obtain a program generator which turns higher-order Scheme programs into first-order tail-recursive Scheme F.

Next, we apply `gen-s` to itself,

$$\texttt{gen-s-ft} = [\![\texttt{gen-s}]\!]_S \ \texttt{gen-s},$$

and obtain the higher-order to F program generator, but now written in F!

Now we start on the C end of the translation. An $F \rightarrow$C compiler (written in C) is the result of an application of `C-mix` to `int-c`:

$$\texttt{gen-c} = [\![\texttt{C-mix}]\!]_C \ \texttt{int-c}.$$

We can now translate `gen-s-ft` to C by using the compiler just constructed:

$$\texttt{gen-s-ft-c} = [\![\texttt{gen-c}]\!]_C \ \texttt{gen-s-ft}.$$

It remains to compose the programs `gen-c` and `gen-s-ft-c` to obtain a full Scheme to C compiler written in C:

$$\texttt{scheme->c} = \texttt{gen-c} \circ \texttt{gen-s-ft-c}$$

Performing the composition merely consists in merging the print routine of `gen-c` with the parser of `gen-s-ft-c`.

In essence, we have seen that a Scheme-to-C compiler (written in C) can be generated by partial evaluation for the price of writing two interpreters, `int-s` and `int-c`:

$$\texttt{scheme->c} = \texttt{int-s} + \texttt{int-c} + \text{partial evaluation.}$$

The ideas presented in this section have not been realized in practice, due to the fact that no C specializer is publicly available as of yet.

6 Experimental Results

We have run some preliminary benchmarks which indicate that the performance of our approach is comparable to other Scheme compilers which generate C code. Our benchmarks are a program computing derivatives `deriv` from the Gabriel benchmark suite [22], the Takeuchi function `tak`, a CPS version of it `cpstak`, a version of it using lists instead of integers `takl` (also taken from the Gabriel suite) a version of the Fibonacci function involving closures `fibclos`, a suite of calls to `cps-append`, and a program solving the 10-queens problem `queens`. Figure 8 shows the timings of the benchmarks as compared with Hobbit 4d [43], an optimizing compiler which produces code for the scm runtime, used with maximum optimization and fixnum arithmetic. Our versions of the benchmarks were all run using the offline generalization strategy. The tests were run on an IBM PowerPC/250.

The `fibclos` and `cps-append` benchmarks indicate that our approach deals especially well with higher-order code. For the first-order code in `tak`, `deriv`, and `queens`, our approach introduces evaluation contexts and thus closures whereas Hobbit can use the native C stack to some advantage. Note that we have spent no effort whatsoever on tuning either the resulting first-order Scheme code, or the translation to C. We believe that further optimizations will result in an additional substantial performance increase. Also, using the online generalization strategy, the `cpstak` benchmark ran roughly 3 times faster.

Our compiler produces quite compact stand-alone executables. The complete benchmark suite yields a binary well under 200 Kilobytes—including the Boehm collector.

The programs associated with the optimizing compiler to tail-recursive Scheme take up less than 70 Kilobytes. The compiler to C takes up a mere 10 Kilobytes.

	We	Hobbit
`deriv`	2420	390
`tak`	5820	810
`cpstak`	6400	6490
`takl`	220	870
`fibclos`	15820	19480
`cps-append`	5480	36340
`queens`	8110	2370

Figure 8: Benchmarks (timings in milliseconds)

7 Related Work

Turchin [47] shows that the interpretive approach can perform powerful transformations. Glück and Jørgensen [24, 25] use the interpretive approach to generate a deforester and a supercompiler. However, they only deal with first-order languages. Past attempts at compilers for higher-order languages by partial evaluation have produced higher-order target code because they are written in higher-order languages. Bondorf [7] studies the automatic generation of a compiler for a lazy higher-order functional language from an interpreter. Jørgensen shows that optimizing compilers for realistic functional languages can be generated by rewriting an interpreter [30, 31]. Consel and Danvy [17] use partial evaluation to compile Algol to tail-recursive Scheme. However, they attribute their success to sophisticated features of the partial evaluator they use, Schism, such as partially static data structures and higher-order functions. Burke and Consel [12] translate Scheme into low-level stack-machine code by multiple interpretive passes, starting from a denotational semantics for Scheme. However, they also make extensive use of higher-order features of the partial evaluator.

The first mention of higher-order removal or defunctionalization appears in work of Reynolds [35]. Compilers for functional languages [42, 4, 3, 20] usually achieve closure conversion with a direct non-optimizing transformation algorithm, and employ CPS conversion to transform programs into tail form. Chin and Darlington [13, 14] give a higher-order removal algorithm for lazy functional languages. However, the resulting program may still be higher-order—the algorithm does not perform closure conversion. The compilation of higher-order languages via a C compiler has been used successfully in several projects, such as sml2c [44], Hobbit [43], Bigloo [39], and the Glasgow Haskell Compiler [34]. In particular, sml2c also translates tail-recursive intermediate code obtained from a CPS transformation, but uses a function dispatcher for handling control.

8 Conclusion

We have used the interpretive approach to generate the middle end of a compiler for a strict, higher-order functional language from an interpreter. We achieve closure conversion and conversion to tail form by applying the respective transformations on a straightforward interpreter manually. Offline partial evaluation turns the interpreter into an automatic transformer by virtue of the language preservation property. Adding constant propagation in static data to our interpreter then turns the simple transformer into an optimizer and specializer thanks to the specializer projections. The translation also makes optimizations present in the partial evaluator such as post-unfolding and arity raising accessible to the optimizied programs. In addition, we have presented a translator of the resulting code into low-level C.

We have formulated the language preservation property, and put it to use for the optimizing compilation of a higher-order language into C with little conceptual effort. We consider this a successful bootstrapping process. Our results prove that partial evaluation is a practical approach to the generation of optimizing compilers.

References

[1] Lars Ole Andersen. Self-applicable C program specialization. In Consel [15], pages 54–61. Report YALEU/DCS/RR-909.

[2] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

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

[4] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, pages 293–302, 1989.

[5] Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors. *Partial Evaluation and Mixed Computation*. North-Holland, 1987. Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation.

[6] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *Proc. of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 157–164, Toronto, June 1991. ACM. SIGPLAN Notices, v26, 6.

[7] Anders Bondorf. Automatic autoprojection of higher order recursive equations. In Jones [27], pages 70–87. LNCS 432.

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

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

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

[11] Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, July 1993.

[12] E. David Burke and Charles Consel. Compiling scheme programs via multi-pass partial evaluation. Technical report, Oregon Graduate Institute of Science and Technology, 1994.

[13] Wei-Ngan Chin. Fully lazy higher-order removal. In Consel [15], pages 38–47. Report YALEU/DCS/RR-909.

[14] Wei-Ngan Chin and John Darlington. Higher-order removal transformation technique for functional programs. In *Proc. of 15th Australian Computer Science Conference*, pages 181–194, Hobart, Tasmania, January 1992. Australian CS Comm Vol 14, No 1.

[15] Charles Consel, editor. *Workshop Partial Evaluation and Semantics-Based Program Manipulation '92*, San Francisco, CA, June 1992. Yale University. Report YALEU/DCS/RR-909.

[16] Charles Consel. A tour of Schism. In *Symp. Partial Evaluation and Semantics-Based Program Manipulation '93*, pages 134–154, Copenhagen, Denmark, June 1993. ACM.

[17] Charles Consel and Olivier Danvy. Static and dynamic semantics processing. In *Symposium on Principles of Programming Languages' 91*, pages 14–24. ACM, January 1991.

[18] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Symposium on Principles of Programming Languages '93*, pages 493–501, Charleston, January 1993. ACM.

[19] *Functional Programming Languages and Computer Architecture*, London, GB, 1989.

[20] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of programming languages*. MIT Press, Cambridge, MA, 1992.

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

[22] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems*. MIT Press, Cambridge, MA, 1985.

[23] Robert Glück. On the generation of specializers. *Journal of Functional Programming*, 4(4):499–514, October 1994.

[24] Robert Glück and Jesper Jørgensen. Generating optimizing specializers. In *IEEE International Conference on Computer Languages*, pages 183–194. IEEE Computer Society Press, 1994.

[25] Robert Glück and Jesper Jørgensen. Generating transformers for deforestation and supercompilation. In B. Le Charlier, editor, *Static Analysis*, volume 864 of *Lecture Notes in Computer Science*, pages 432–448. Springer-Verlag, 1994.

[26] Neil D. Jones. Challenging problems in partial evaluation and mixed computation. In Bjørner et al. [5], pages 1–14. Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation.

[27] Neil D. Jones, editor. *Proc. of the 3rd European Symposium on Programming 1990*, Copenhagen, Denmark, 1990. Springer-Verlag. LNCS 432.

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

[29] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, pages 124–140, Dijon, France, 1985. Springer-Verlag. LNCS 202.

[30] Jesper Jørgensen. Compiler generation by partial evaluation. Master's thesis, DIKU, University of Copenhagen, 1991.

[31] Jesper Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *Symposium on Principles of Programming Languages '92*, pages 258–268. ACM, ACM, January 1992.

[32] John Launchbury. *Projection Factorisations in Partial Evaluation*, volume 1 of *Distinguished Dissertations in Computer Science*. Cambridge University Press, 1991.

[33] Torben Æ. Mogensen. Separating binding times in language specifications. In FPCA1989 [19], pages 14–25.

[34] Simon L Peyton Jones, Cordelia Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell compiler: a technical overview. In *Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference*, Keele, 1993.

[35] John C. Reynolds. Definitional interpreters for higher-order programming. In *ACM Annual Conference*, pages 717–740, July 1972.

[36] Sergei A. Romanenko. A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In Bjørner et al. [5], pages 445–464. Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation.

[37] Sergei A. Romanenko. Arity raiser and its use in program specialization. In Jones [27], pages 341–360. LNCS 432.

[38] Erik Ruf. *Topics in Online Partial Evaluation*. PhD thesis, Stanford University, Stanford, CA 94305-4055, March 1993. Technical report CSL-TR-93-563.

[39] Manual Serrano. Bigloo user's manual. Technical report, INRIA, 1994. (to appear).

[40] Peter Sestoft. Replacing function parameters by global variables. In FPCA1989 [19], pages 39–53.

[41] Richard M. Stallman. *Using GNU CC*, November 1995. (part of the GCC distribution).

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

[43] Tanel Tammet. Lambda-lifting as an optimization for compiling scheme to C. available as `ftp://www.cs.chalmers.edu/pub/users/tammet/www/hobbit.ps`.

[44] D. Tarditi, A. Acharya, and P. Lee. No assembly required: compiling Standard ML to C. Technical Report CMU-CS-90-187, School of Computer Science, Carnegie Mellon University, November 1990.

[45] Peter Thiemann. Higher-order redundancy elimination. In Peter Sestoft and Harald Søndergaard, editors, *Workshop Partial Evaluation and Semantics-Based Program Manipulation '94*, pages 73–84, Orlando, Fla., June 1994. ACM.

[46] Peter Thiemann and Robert Glück. The generation of a higher-order online partial evaluator. In Masato Takeichi, editor, *Fuji Workshop on Functional and Logic Programming*, pages 239–253, Fuji Susono, Japan, July 1995. World Scientific Press, Singapore.

[47] Valentin F. Turchin. Program tranformation with metasystem transitions. *Journal of Functional Programming*, 3(3):283–313, July 1993.

[48] Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic online partial evaluation. In *Conf. Functional Programming Languages and Computer Architecture '91*, pages 165–191, Cambridge, September 1991. ACM.