Bootstrapping Higher-Order Program Transformers from Interpreters

Michael Sperber* Universität Tübingen
Robert Glück† University of Copenhagen
Peter Thiemann[1] Universität Tübingen

Keywords: meta programming, higher-order programming, program transformation, partial evaluation

Abstract

Partial evaluation can automatically generate program transformers from interpreters. In the context of functional languages, we investigate the design space of higher-order interpreters to achieve certain transformation effects. Our work is based on the interpretive approach and exploits the language preservation property of offline partial evaluators.

We have generated higher-order online partial evaluators, optimizing closure converters, and converters to first-order tail form. The latter can serve as the middle end of a compiler. The generated transformers are strictly more powerful than the partial evaluators used for their generation.

1 Introduction

Specialization of a self-applicable partial evaluator with respect to an interpreter produces a compiler. This paper investigates the generation of *optimizing program transformers* from interpreters. We study different types of interpreters for a strict, higher-order functional language, and show how to use different partial evaluators to achieve closure conversion, higher-order removal, conversion into tail-recursive form, and online specialization. Given an appropriate interpreter, the *interpretive approach* can drastically change the result of the transformation [32, 18]. Our interpreters exploit two basic principles: *self-application* to generate stand-alone transformers, and the *language-preservation property* of offline partial evaluators to transform higher-order constructs.

Using these principles, we have bootstrapped an online specializer for a higher-order language from an offline partial evaluator for a first-order language. Writing such a specializer by hand is a significant undertaking [29, 33]. Moreover, our transformer to first-order tail form is an optimizing version of Reynolds's defunctionalization [27].

This contribution extends previous work on the generation of program transformers by partial evaluation in several directions. Whereas previously published experiments deal with first-order programs, our transformers handle a higher-order subset of Scheme:

1. A constant-propagating interpreter written in higher-order Scheme using partially static data structures yields an online specializer for a higher-order language.

2. An interpreter written in first-order Scheme using partially static data structures results in a converter of higher-order programs into first-order programs; it performs higher-order removal and closure conversion.

3. An interpreter written in a first-order, tail-recursive subset of Scheme that propagates constants produces an online specializer for a higher-order language. It generates first-order, tail-recursive residual programs which are easy to compile to native code [13].

Overview Section 2 reviews partial evaluation. Section 3 introduces the *specializer projections* and the *language preservation property* which are crucial to the bootstrapping process. In Sec. 4, we describe the implementation techniques common to all our interpreters, and Sec. 5 illustrates the different transformation effects. Section 6 presents the interpreters in more detail, and Sec. 7 discusses related work.

2 Background: Flavors of Partial Evaluation

Partial evaluation is an automatic program transformation that performs aggressive constant propagation: if parts of the input of a *subject program* are known at compile time, a partial evaluator propagates this *static* input to generate a specialized *residual program*. The residual program takes the remaining, *dynamic* parts of the input as parameters and produces the same results as the subject program applied to the complete input. Partial evaluation can remove interpretive overhead, and thereby produce significant speed-ups [21].

In this work, we deal with *offline* and *online* partial evaluation. An offline partial evaluator consists of a *binding-time analysis* and a reducer. The binding-time analysis, given the subject program and the binding time of its arguments, annotates each expression in the program with a binding time, "static" or "dynamic." The reducer processes the annotated

* Address: Wilhelm-Schickard-Institut, Universität Tübingen, Sand 13, 72076 Tübingen, Germany. {sperber,thiemann}@informatik.uni-tuebingen.de

† Address: DIKU, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark. glueck@diku.dk

program and the static part of the input. Driven by the annotations, it reduces static expressions and rebuilds dynamic ones.

An *online partial evaluator* decides "online" whether to reduce or rebuild an expression, based on the static components of actual values of the subexpressions. Online partial evaluators are 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.

In our discussion, we describe the effects of a progression of features of offline partial evaluators:

Simple partial evaluators treat all data as either completely static or dynamic. Examples are the *Scheme0* specializer of [21], *Unmix*, a descendant of the Moscow specializer [28], and early versions of *Similix* [5] and *Schism* [9].

If only parts of data structures are dynamic, partial evaluators should respect such *partially static data*. Several binding-time analyses and reducers handle first-order languages with partially static data [25, 10, 24].

Current partial evaluators such as Similix [4] and Schism [11] support higher-order languages.

3 Basic Principles

To achieve the transformation effects, we exploit two principles: *self-application* to generate stand-alone transformers from interpreters, and the *language-preservation property* of offline partial evaluators as the basis for higher-order removal and conversion to tail form.

Transformer Generation Partial evaluation of interpreters can generate program transformers [16], a technique called the *interpretive approach* [18].

It can also perform compilation: The specification of an L-interpreter `int` written in S is

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

where $[\![\_]\!]_L$ denotes the execution of an L-program, $\texttt{p}_L$ is an L-program, and `inp` is its input. An $S \to S$-partial evaluator `pe` written in S can compile $\texttt{p}_L$ into an equivalent S-program $\texttt{p}_S$ by the first *Futamura projection* [14]:

$$\texttt{p}_S = [\![\texttt{pe}]\!]_S\ \texttt{int}^{sd}\ \texttt{p}_L$$

The sd superscript of `int` indicates that `pe` 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 [21].

A generalization of the Futamura projections shows how to generate a specializer from an interpreter. The key idea is to use a *two-level interpreter* [18, 18] `2int` which accepts the input to the interpreted program in a static and a dynamic component. The interpreter tries to perform each operation with the static component of the input first; only if this fails, the dynamic component is consulted. Specialized programs result from the first *specializer projection* [16]:

$$\texttt{r}_S = [\![\texttt{pe}]\!]_S\ \texttt{2int}^{ssd}\ \texttt{p}_L\ \texttt{inp}_s.$$

where $\texttt{inp}_s$ is the static part of the input and $\texttt{r}_S$ is the specialized program. Analogous to compiler generation, self-application of the partial evaluator generates stand-alone specializers and specializer generators.

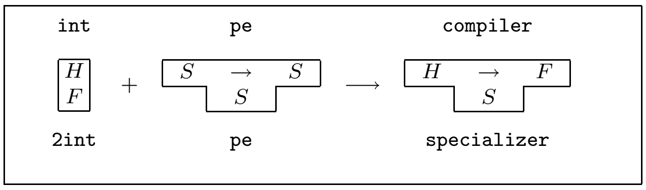

Figure 1: Compiler and specializer generation ($F \subseteq S$)

Language Preservation Offline partial evaluators have the *language preservation property:* for any sublanguage $S' \subseteq S$ which includes all constants, and for any binding-time annotated S-program `p` every dynamic expression of which belongs to S', $[\![\texttt{pe}]\!]\ \texttt{p x} \in S'$ holds for arbitrary static `x`. This property can be verified by inspecting the specialization phase of an offline partial evaluator [21].

Let `pe` be a language-preserving $S \to S$-partial evaluator, and let `int` be an interpreter for a higher-order language H written in a first-order language $F \subseteq S$. Specializing the H-interpreter `int` with respect to a H-program `p` translates an H-program into an F-program. Figure 1 illustrates this.

Because `pe` preserves the F-ness of the subject program, the residual programs $\texttt{p}_F = [\![\texttt{pe}]\!]\ \texttt{int}^{sd}\ \texttt{p}_H$ (compiled program) and $\texttt{r}_F = [\![\texttt{pe}]\!]\ \texttt{2int}^{ssd}\ \texttt{p}_H\ \texttt{inp}$ (specialized program) are F-programs.

4 Higher-Order Interpreters for Partial Evaluation

Some general remarks apply to all interpreters described in the next sections. All are recursive-descent interpreters, treat the same subject language, and use a common flow analysis.

$E \in$ Expr, $D \in$ Definition, $\Pi \in$ Program

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

Figure 2: Syntax

Our interpreters treat a higher-order, side-effect free subset of Scheme [20]. The syntax is defined in Fig. 2. V denotes variables, K constants, O primitive operators. We assume. without loss of generality, that `lambda` and `let` expressions abstract one variable and that defined functions occur only in applications.

Fig. 3 shows the ancestor of our interpreters. The metalanguage is a call-by-value lambda calculus enriched with constants, sums, and products. The notation Value* $\to$ Value abbreviates the sum of $()$ $\to$ Value, Value $\to$ Value, Value $\times$ Value $\to$ Value, etc. We have omitted the injection tags and case analysis for the elements of Value.

Exploiting Flow Information Most optimizing transformations for higher-order programs depend on flow information to do their work. Similarly, our interpreters perform an equational flow analysis [6] on the input program prior to interpretation proper. After assigning a flow-variable to each expression and binding occurrence of a variable, the flow analysis partitions the flow variables into *flow classes*. Each flow class consists of the flow variables of expressions the values of which may flow together during execution.

$$\begin{array}{rcl}
\mathsf{Value} & = & \mathsf{BaseValue} + \mathsf{Value} \to \mathsf{Value} \\
\rho \in \mathsf{Env} & = & \mathsf{Var} \to \mathsf{Value} \\
\psi \in \mathsf{ProcEnv} & = & \mathsf{Procedure} \to \mathsf{Expr} \\
\mathcal{K}[\![\_]\!] & : & \mathsf{Constants} \to \mathsf{Value} \\
\mathcal{O}[\![\_]\!] & : & \mathsf{Operators} \to \mathsf{Value}^* \to \mathsf{Value} \\
\mathcal{E}[\![\_]\!] & : & \mathsf{Expr} \to \mathsf{ProcEnv} \to \mathsf{Env} \to \mathsf{Value}
\end{array}$$

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

Figure 3: A standard call-by-value interpreter

In the closure-converting interpreters, the interpreter must represent some closures as dynamic data structures. The flow information comes into play when a dynamic closure reaches an application. The interpreter tests it against all `lambda`s which could have generated the closure by traversing the corresponding flow class. Thanks to this binding-time improvement, *The Trick* [21], it is possible to keep the expression argument to the interpreter.

The closure-converting interpreters use flow information to compute which closures they need to represent by dynamic data, so as to ensure termination.

5 Dimensions of Program Transformation

We illustrate the different transformation effects achieved by partial evaluation of the three different interpreters with respect to the same example program—a version of a list append procedure written in continuation-passing style.

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

Our interpreters transform the above program in several different ways, performing specialization and closure conversion to varying degrees. All the examples have in common that constructs that the interpreter handles in the same way as the straightforward interpreter in Fig. 3 are merely transliterated from subject to residual program. Wherever the interpreter uses nonstandard techniques, such as constant propagation or closure passing, actual transformations take place.

Higher-Order Online Partial Evaluation Specializing the constant-propagating, higher-order interpreter (Sec. 6.1) with respect to the `append` program, with `x` $\mapsto$ `(foo bar)` static, and `y` dynamic, performs online specialization, resulting in:

```
(define (s1-2int-skeleton-0 xd*_0)
   (cons 'foo (cons 'bar (car xd*_0))))
```

Higher-Order Removal Specializing the first-order, higher-order-removing interpreter (Sec. 6.2) with respect to the `append` program and dynamic input produces a residual program with explicit closure passing. Closures are represented as lists $(\ell\ .\ v^*)$ of a closure label ℓ and the values v^* of its free variables. Label `4` denotes the identity, `17` the inner continuation.

```
(define (s1-int-0 x*_0)
   (define (s1-eval-0-1 a*_0 a*_1 a*_2)
     (if (null? a*_2)
         (loop-0-2 (car a*_0) a*_0 a*_1)
         (s1-eval-0-1 (list 17 a*_0 a*_2) a*_1 (cdr a*_2))))
   (define (loop-0-2 dyn-nr_0 dyn-cl_1 rand_2)
     (if (equal? 4 dyn-nr_0)
         rand_2
         (let* ((dd_4 (cdr dyn-cl_1)) (g_5 (car dd_4)))
           (loop-0-2
             (car g_5)
             g_5
             (cons (car (car (cdr dd_4))) rand_2)))))
   (s1-eval-0-1 (list 4) (car (cdr x*_0)) (car x*_0)))
```

Higher-Order-Removing Partial Evaluation Specializing the constant-propagating first-order tail-recursive interpreter (Sec. 6.3) treats the continuation as dynamic. Therefore, explicit closures appear in the residual code. However, if the first argument `x` is the static list `(foo bar)`, the interpreter encodes the list in the closure representation, eliminating the `null?` test:

```
(define (s1-int-$1 cv-vals)
  (s1-eval-$8 (car cv-vals)
              '(17 (4) (foo bar))
              '(bar)))

(define (s1-eval-$8 cv-vals-$1 cv-vals-$2 cv-vals-$3)
  (if (equal? 4 (car cv-vals-$2))
      (cons (car cv-vals-$3) cv-vals-$1)
      (s1-eval-$8 (cons (car cv-vals-$3) cv-vals-$1)
                  (cadr cv-vals-$2)
                  (caddr cv-vals-$2))))
```

6 Bootstrapping Transformers from Interpreters

In this section, we describe the interpreters used to generate the program transformers. They demonstrate a trade-off between the versatility of the partial evaluator and the complexity of the interpreter. As we ignore features of the partial evaluator or use a weaker one, the interpreters get more complex.

6.1 Bootstrapping a Higher-Order Online Specializer

In our first application we use the specializer projections to generate a $H \to H$-online specializer by specializing a two-level H-interpreter. We use the offline partial evaluator Similix [4] which handles a higher-order dialect of Scheme with partially static algebraic data types.

The choice of an appropriate data representation is crucial for effective specialization. The interpreter uses the type

$$\begin{array}{rcl}
\mathit{Data} & ::= & \texttt{Atom}\ (\mathit{SchemeValue}) \\
 & | & \texttt{Cons}\ (\mathit{Data} \times \mathit{Data}) \\
 & | & \texttt{Closure}\ (\mathit{Label} \times \mathit{Data}^*) \\
 & | & \texttt{Value}\ (\mathit{SchemeValue})
\end{array}$$

The interpreter propagates *Data* values such that the constructor is always static. The binding times of the components are independent of each other.

The arguments of `Atom` are static first-order Scheme values. `Cons` is for data structures. `Closure` objects represent partially static closures with the label of a lambda abstraction and a list of the values of the free variables. The argument of `Value` is an arbitrary Scheme value, first-order or higher-order.

`Atom` marks a fully static value whereas `Value` marks a fully dynamic value. `Cons` and `Closure` construct partially static data. The interpreter uses a static conversion function to transform arbitrary data into a dynamic `Value` object. This is different to offline partial evaluators which only lift first-order values.

Propagating information of *Data* values creates two problems for partial evaluation:

Loss of constructor information The binding-time analyses used in most offline partial evaluators treat the result of a *dynamic conditional* (`if` E_0 E_1 E_2) with E_0 dynamic as fully dynamic, and do not propagate constructor information from the branches. Therefore, the `Value/Closure` distinction is lost in a dynamic conditional, which in turn forces the binding-time analysis to treat all values in the interpreter as dynamic, and forfeits all optimization. Consequently, the interpreter needs to "re-attach" (eta-expand) the constructor information to the result of a dynamic `if`:

```
(Value (if (get-Value $\mathcal{E}[\![E_0]\!]\psi\rho$)
           (get-Value $\mathcal{E}[\![E_1]\!]\psi\rho$)
           (get-Value $\mathcal{E}[\![E_2]\!]\psi\rho$)))
```

`Get-Value` extracts the value from a `Value` object. As the above code assumes that the result of the branches is a `Value` object, the interpreter must coerce all values that might be the result of a dynamic conditional into `Value` objects, thereby *dynamizing* them.

Nontermination If a subject program has parameters that grow statically under dynamic control, the transformation does not terminate as is. A prominent example is the continuation parameter of a recursive function written in continuation-passing style where the recursion is under dynamic control. The interpreter needs to detect these situations and dynamize the accumulating parameters.

To achieve the dynamization of accumulating data when necessary, the interpreter dynamizes all values in the current environment when it encounters a dynamic conditional or dynamic `lambda` [5], thereby removing them from the view of the partial evaluator and causing it to insert a *memoization point* which results in a specialized residual function which can be re-used, thereby avoiding infinite unfolding.

6.2 Bootstrapping a Higher-Order Remover

Our second interpreter exploits the language-preservation property to convert a higher-order program into a first-order program. The interpreter uses partially static data structures, but no higher-order functions. Agin, we use Similix.

The interpreter represents higher-order functions by closures. When evaluating a `lambda` expression, it produces a list containing the flow variable (closure label) of the `lambda` and the values of its free variables. An application of a dynamic closure finds the `lambda` expression of the closure, creates a new environment from the values of the free variables, and continues interpretation with the `lambda` body.

$$\begin{array}{rcl} \text{Value} & = & \text{BaseValue} + \text{Label} \times \text{Var} \times \text{Env} \\ \psi \in \ \text{ProcEnv} & = & (\text{Procedure} + \text{Label}) \rightarrow \text{Expr} \end{array}$$

$$\begin{array}{rl} \mathcal{E}[\![(\texttt{lambda}^{\ell}\ (V)\ E)]\!]\psi\rho = & (\ell, V, \rho) \\ \mathcal{E}[\![(E_1\ E_2)]\!]\psi\rho = & \mathit{let}\ (\ell, V, \rho') = \mathcal{E}[\![E_1]\!]\psi\rho \\ & \mathit{in}\ \mathcal{E}[\![\psi(\ell)]\!]\rho'[V \mapsto \mathcal{E}[\![E_2]\!]\psi\rho] \end{array}$$

Figure 4: Standard interpreter after closure conversion

Straightforward application of this approach treats all closure representations as dynamic. Whenever an application is reached, an instance of The Trick described in Sec. 4 loops over the `lambda` bodies that could possibly reach the application, and continues evaluation using the body found.

It is possible to avoid building many closures at run time by keeping them static, exploiting the constant propagation of the partial evaluation process to propagate lambdas through the program. This achieves actual higher-order removal [7]. To distinguish between closures propagated during transformation ("static") and those created at run time ("dynamic"), the interpreter uses a rudimentary two-level representation which is a reduced version of the data representation of the online specializer in Sec. 6.1

$$\mathit{Data} ::= \texttt{Value}\ (\mathit{SchemeValue}) \mid \texttt{Closure}\ (\mathit{Label} \times \mathit{Data}^*)$$

Since, for this interpreter, all conditionals are dynamic, the dynamization strategy of the online specializer is not applicable here. Instead, the dynamization is controlled by an offline *dynamization analysis* which marks all `lambda` expressions which must evaluate to dynamic closures. It uses the results of the flow analysis. When the interpreter avoids the above problems as described, specialization always terminates.

6.3 Bootstrapping a Higher-Order-Removing Online Specializer

The language-preservation property applies equally well to two-level interpreters, and the specializers generated from them: if the two-level interpreter is written in first-order, tail-recursive style, so are the residual programs. Thus, the resulting transformer performs higher-order removal and specialization in one pass.

Now, we use Unmix to specialize the interpreter. As Unmix has neither partially static data structures nor higher-order functions, the interpreter turns a simple-minded first-order offline partial evaluator into a much more powerful higher-order online specializer which supports partially static data structures.

We have extended the higher-order removal interpreter to a two-level interpreter, and encoded partially static data structures by splitting them into a completely static and a completely dynamic part. We changed the closure-converting interpreter as follows.

Completely static *value descriptions* (`quote-desc` K) take the part of values. Since the specializer must handle partially static data, the dynamic components of value descriptions are references to *configuration variables* (`cv-desc` *int*) the values of which are stored in a separate environment. Value descriptions have the following form:

$$\begin{array}{rcl} \mathit{desc} & ::= & (\texttt{quote-desc}\ K) \\ & | & (\texttt{cons-desc}\ \mathit{desc}\ \mathit{desc}) \\ & | & (\texttt{closure-desc}\ \ell\ \mathit{desc}^*) \\ & | & (\texttt{cv-desc}\ \mathit{int}) \end{array}$$

Our approach extends that of Glück and Jørgensen [18] by closures. Configuration variables are numbered. Consequently, the environment consists of three components: (i) a static program variable environment mapping program variable identifiers to value descriptions, (ii) a static list of configuration variable numbers currently in use, and (iii) a dynamic list of the values of the configuration variables.

Since the return value of the interpretation function is dynamic, the data structure information that is supposed to be present in the value description is lost when `cons` is treated as a normal primitive. To preserve the static parts of data structures, it is necessary to treat those expressions specially which the interpreter can translate directly into value descriptions—the *simple expressions*:

$$SE ::= V \mid K \mid \texttt{(}O\ SE^{*}\texttt{)} \mid \texttt{(lambda (}V\texttt{)}\ E\texttt{)}$$

The translation of simple expressions into descriptions is straightforward and static.

The syntax of the input language of the two-level interpreter allows only simple expressions as arguments to procedure calls, as operands to applications, as conditional tests, and as operands to `car`, `cdr`, and `cons`. A desugaring phase before the actual interpretation converts programs into the restricted syntax, inserting `lets` to pull non-simple expressions out of contexts where they are not allowed. With these changes, the interpreter preserves the structure of partially static data.

With the introduction of simple expressions, the burden of actual evaluation gets placed on `let` which allows general expressions in both positions. The naive way to handle `let` generalizes the bound expression and puts it in the configuration variable environment. A better way is to extend the approach of [18]: when a $\texttt{(let ((}V\ E_1\texttt{))}\ E_2\texttt{)}$ is encountered, the variable V and E_2 are put on a stack of pending `lets`, along with the current program variable environment. Evaluation proceeds with E_1. When the interpreter returns from a simple expression, it checks for a pending `let`, and, if there is one, performs the binding continuing with E_2. Otherwise, interpretation is complete.

To prevent the configuration variable environment from growing without bounds, the interpreter normalizes it when the set of accessible configuration variables changes, that is, when the interpreter applies a dynamic closure, and when it pops the stack of pending `lets`. Only the accessible configuration variables remain and are renumbered canonically.

With these changes, the interpreter performs online specialization. As a side effect of the introduction of simple expressions, the interpreter never calls itself in a non-tail position. In all places where non-tail calls occur in the original interpreter, the restricted syntax allows only simple expressions which can be converted to value descriptions statically.

7 Related Work

The only online partial evaluator for a realistic, higher-order, functional language we know of is Fuse by Ruf [29], Weise, and others [33]. Mogensen reports an online partial evaluator for the lambda calculus [26]. Other online approaches that go beyond constant propagation in a first-order setting are supercompilation [31, 19], and generalized partial computation [15].

Our work extends previous work that uses the interpretive approach to achieve transformation effects. Turchin [32] shows that the interpretive approach makes transformations possible which the direct application of supercompilation to a subject program cannot perform. The generation of specializers and the bootstrapping process is discussed by Glück [16]. Glück and Jørgensen [18, 17] use the interpretive approach to generate Wadler's deforestation algorithm and a version of Turchin's supercompiler using an offline partial evaluator. However, they treat only first-order languages.

Past attempts at compilers for higher-order languages generated by partial evaluation have always produced higher-order target code because the interpreters were written in higher-order languages. Bondorf [3] 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 [22, 23]. Consel and Danvy [12] use partial evaluation to compile Algol to tail-recursive Scheme. They attribute their success to sophisticated features of their partial evaluator, Schism, such as partially static data structures, combinator extraction, and higher-order functions.

The first mention of higher-order removal or defunctionalization is due to Reynolds [27]. Compilers for functional languages [2, 1, 13] achieve closure conversion with a manually developed transformation algorithm, not by specialization. Ordinary closure conversion does not create specialized versions of higher-order functions and does not look across function boundaries.

Chin and Darlington [7, 8] give a higher-order removal algorithm for lazy functional languages. However, the resulting program may still be higher-order and closure conversion is not the aim of the algorithm.

8 Conclusion

We have used interpreters for a strict, higher-order functional language to perform a variety of useful program transformation tasks, among them optimizing closure conversion, conversion to tail form, and online specialization. Whereas previously published experiments deal with first-order programs, our transformers handle a higher-order subset of Scheme. This paper reports the automatic generation of higher-order online specializers, and of optimizing higher-order removers.

The more powerful the partial evaluator, the easier it is to write the corresponding interpreters. The presence of partially static data structures and higher-order partial evaluation, independently, simplifies the interpreters, but they are not necessary.

It is also remarkable that the generated specializers are more powerful than the partial evaluators used to generate them. One single interpreter specialized with a simple first-order partial evaluator without partially static data structures can generate a significantly more powerful online specializer which performs aggressive constant propagation, higher-order removal, and conversion to tail form.

Moreover, we used a partial evaluator for a first-order language to *bootstrap* a partial evaluator for a higher-order language. The original higher-order interpreter was written in a first-order subset of Scheme and converted into a partial evaluator according to the specializer projections. It is easy to imagine similar interpreters for other higher-order languages.

Generating the transformers from interpreters is simpler than writing the transformers directly. The underlying partial evaluator optimizes information propagation through the program if the the interpreter is suitably written. It remains to choose appropriate dynamization strategies to prevent non-termination.

References

[1] APPEL, A. W. *Compiling with Continuations*. Cambridge University Press, 1992.

[2] APPEL, A. W., AND JIM, T. Continuation-passing, closure-passing style. In *Proc. 16th Annual ACM Symposium on Principles of Programming Languages* (Austin, Texas, Jan. 1989), ACM Press, pp. 293–302.

[3] BONDORF, A. *Self-applicable partial evaluation*. PhD thesis, DIKU, University of Copenhagen, 1990. DIKU Report 90/17.

[4] BONDORF, A. *Similix 5.0 Manual*. DIKU, University of Copenhagen, May 1993.

[5] BONDORF, A., AND DANVY, O. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming 16*, 2 (1991), 151–195.

[6] BONDORF, A., AND JØRGENSEN, J. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming 3*, 3 (July 1993), 315–346.

[7] CHIN, W.-N. Fully lazy higher-order removal. In *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM '92* (San Francisco, CA, June 1992), C. Consel, Ed., Yale University, pp. 38–47. Report YALEU/DCS/RR-909.

[8] CHIN, W.-N., AND DARLINGTON, J. Higher-order removal transformation technique for functional programs. In *Proc. of 15th Australian Computer Science Conference* (Hobart, Tasmania, Jan. 1992), pp. 181–194. Australian CS Comm Vol 14, No 1.

[9] CONSEL, C. New insights into partial evaluation: the Schism experiment. In *ESOP '88* (Nancy, France, 1988), H. Ganzinger, Ed., vol. 300 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 236–246.

[10] CONSEL, C. Binding time analysis for higher order untyped functional languages. In *Proc. 1990 ACM Conference on Lisp and Functional Programming* (Nice, France, 1990), ACM Press, pp. 264–272.

[11] CONSEL, C. A tour of Schism. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '93* (Copenhagen, Denmark, June 1993), D. Schmidt, Ed., ACM Press, pp. 134–154.

[12] CONSEL, C., AND DANVY, O. Static and dynamic semantics processing. In *Proc. 18th Annual ACM Symposium on Principles of Programming Languages* (Orlando, Florida, Jan. 1991), ACM Press, pp. 14–24.

[13] FRIEDMAN, D. P., WAND, M., AND HAYNES, C. T. *Essentials of Programming Languages*. MIT Press and McGraw-Hill, 1992.

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

[15] FUTAMURA, Y., NOGI, K., AND TAKANO, A. Essence of generalized partial computation. *Theoretical Comput. Sci. 90*, 1 (1991), 61–79.

[16] GLÜCK, R. On the generation of specializers. *Journal of Functional Programming 4*, 4 (Oct. 1994), 499–514.

[17] GLÜCK, R., AND JØRGENSEN, J. Generating optimizing specializers. In *IEEE International Conference on Computer Languages 1994* (Toulouse, France, 1994), IEEE Computer Society Press, pp. 183–194.

[18] GLÜCK, R., AND JØRGENSEN, J. Generating transformers for deforestation and supercompilation. In *Static Analysis* (1994), B. Le Charlier, Ed., vol. 864 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 432–448.

[19] GLÜCK, R., AND KLIMOV, A. V. Occam's razor in metacomputation: the notion of a perfect process tree. In *Static Analysis* (Padova, Italia, Sept. 1993), G. Filé, Ed., vol. 724 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 112–123.

[20] IEEE. Standard for the Scheme programming language. Tech. Rep. 1178-1990, Institute of Electrical and Electronic Engineers, Inc., New York, 1991.

[21] JONES, N. D., GOMARD, C. K., AND SESTOFT, P. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[22] JØRGENSEN, J. Compiler generation by partial evaluation. Master's thesis, DIKU, University of Copenhagen, 1991.

[23] JØRGENSEN, J. Generating a compiler for a lazy language by partial evaluation. In *Proc. 19th Annual ACM Symposium on Principles of Programming Languages* (Albuquerque, New Mexico, Jan. 1992), ACM Press, pp. 258–268.

[24] LAUNCHBURY, J. *Projection Factorisations in Partial Evaluation*, vol. 1 of *Distinguished Dissertations in Computer Science*. Cambridge University Press, 1991.

[25] MOGENSEN, T. . Separating binding times in language specifications. In *Proc. Functional Programming Languages and Computer Architecture 1989* (London, GB, 1989), pp. 14–25.

[26] MOGENSEN, T. . Self-applicable online partial evaluation of pure lambda calculus. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '95* (La Jolla, CA, June 1995), W. Scherlis, Ed., ACM Press, pp. 39–44.

[27] REYNOLDS, J. C. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference* (July 1972), pp. 717–740.

[28] ROMANENKO, S. A. A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In *Partial Evaluation and Mixed Computation* (1987), D. Bjørner, A. P. Ershov, and N. D. Jones, Eds., North-Holland, pp. 445–464. Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation.

[29] RUF, E. *Topics in Online Partial Evaluation*. PhD thesis, Stanford University, Stanford, CA 94305-4055, Mar. 1993. Technical report CSL-TR-93-563.

[30] THIEMANN, P., AND GLÜCK, R. The generation of a higher-order online partial evaluator. In *Fuji International Workshop on Functional and Logic Programming* (Nov. 1995), M. Takeichi and T. Ida, Eds., World Scientific, pp. 239–253.

[31] TURCHIN, V. F. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems 8*, 3 (July 1986), 292–325.

[32] TURCHIN, V. F. Program tranformation with metasystem transitions. *Journal of Functional Programming 3*, 3 (July 1993), 283–313.

[33] WEISE, D., CONYBEARE, R., RUF, E., AND SELIGMAN, S. Automatic online partial evaluation. In *Proc. Functional Programming Languages and Computer Architecture 1991* (Cambridge, MA, 1991), J. Hughes, Ed., no. 523 in Lecture Notes in Computer Science, Springer-Verlag, pp. 165–191.