Bootstrapping Higher-Order Program Transformers from Interpreters

Michael Sperber* Universität Tübingen
Robert Glück† DIKU, University of Copenhagen
Peter Thiemann* Universität Tübingen

Keywords: functional programming, partial evaluation, program transformation.

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. The present 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. A simple-minded partial evaluator for a first-order language suffices to generate the above transformers.

Self-application of a partial evaluator with respect to an interpreter produces a compiler. We investigate the generatation 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* may drastically change the result of the transformation. 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, 32]. 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. Table 1 summarizes our results.

In Sec. 1, we introduce partial evaluation. Section 2 is a tour through the transformers as applied to a single example program. 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. Sec. 5 presents the interpreters in more detail and Sec. 6 discusses related work.

1 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 which only takes the remaining, *dynamic* parts of the input as parameters produces the same results as the subject program applied to the full 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, starting from 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. Guided by the annotations, it reduced static expressions and rebuilds dynamic ones.

An *online partial evaluator* decides "online" in one pass 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 below, we describe the effects of a progression of features of offline partial evaluators.

Simple partial evaluators for first-order languages 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 (*e.g.*, [25, 24]).

Current partial evaluators support higher-order languages (*e.g.*, Similix [4] and Schism [10]).

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

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

higher-order interpreter written in ...	generated transformer
higher-order Scheme using partially static data structures	online specializer for a higher-order language; performs higher-order removal
first-order Scheme using partially static data structures	converter of higher-order programs into first-order programs; performs higher-order removal and closure conversion
first-order, tail-recursive subset of Scheme which propagates constants	online specializer for a higher-order language; generates first-order, tail-recursive residual programs which are easy to compile to native code [13]

Table 1: Transformers generated from interpreters

2 Dimensions of Program Transformation

We illustrate the different transformation effects achieved by partial evaluation of 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))))))
```

Where the interpreter employs straightforward code for the interpretation of a source construct, the residual code exhibits the original structure. Wherever the interpreter uses nonstandard techniques, such as constant propagation or closure passing, the residual code changes accordingly.

Higher-Order Online Partial Evaluation Specializing the higher-order interpreter (Sec. 5.1) with respect to the `append` program, `x` $\mapsto$ `(foo bar)` static, and `y` dynamic produces:

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

Higher-Order Removal Specializing the higher-order removal interpreter (Sec. 5.2) with respect to the `append` program and dynamic input results in a residual program with explicit closure passing. Closures are represented as lists (ℓ . v^*) of a closure label and the values 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 higher-order removal interpreter (Sec. 5.3) 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))))
```

3 Basic Principles

To achieve the effects shown above, 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]. This approach is called the *interpretive approach*.

Partial evaluation of interpreters has received special attention because it can perform compilation. The specification of an L-interpreter `int` written in S is

$$[\![\mathtt{int}]\!]_S\ P_L\ inp = [\![P_L]\!]_L\ inp$$

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

$$P_S = [\![\mathtt{pe}]\!]_S\ \mathtt{int}^{sd}\ P_L$$

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 [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* [17, 19] `2int`

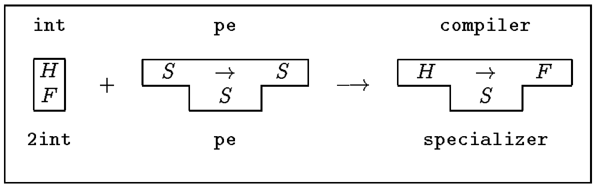

Figure 1: Compiler and specializer generation

which accepts the input to the interpreted program in two components: a static component 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. Residual programs result from the first *specializer projection* [16, 19]:

$$R_S = [\![\texttt{pe}]\!]_S\ \texttt{2int}^{ssd}\ P_L\ inp_s.$$

where inp_s is the static part of the input and R_S is the residual program. Analogous to compiler generation, self-application of the partial evaluator generates stand-alone specializers and specializer generators.

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', $[\![\texttt{pe}]\!]\ P\ x \in S'$ holds for arbitrary static x. This property can be verified by inspecting the specialization phase of an offline partial evaluator, *e.g.*, [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 P into an F-program. Figure 1 illustrates this.

Because `pe` preserves the F-ness of the subject program, the residual programs $P_F = [\![\texttt{pe}]\!]\ \texttt{int}^{sd}\ P_H$ (compiled program) and $R_F = [\![\texttt{pe}]\!]\ \texttt{2int}^{ssd}\ P_H\ inp$ (specialized program) are F-programs as argued before.

4 Higher-Order Interpreters for Partial Evaluation

Some general remarks apply to the higher-order interpreters described in the next section. All are recursive-descent interpreters, treat the same subject language, and use the same flow analysis.

$$\begin{array}{l}
E \in \mathsf{Expr}, D \in \mathsf{Definition}, \Pi \in \mathsf{Program} \\
E ::= V \mid K \mid \texttt{(if}\ E\ E\ E\texttt{)} \mid \texttt{(}O\ E^*\texttt{)} \mid \texttt{(}P\ E^*\texttt{)} \mid \\
\qquad \texttt{(let ((}V\ E\texttt{))}\ E\texttt{)} \mid \texttt{(lambda (}V\texttt{)}\ E\texttt{)} \mid \texttt{(}E\ E\texttt{)} \\
D ::= \texttt{(define (}P\ V^*\texttt{)}\ E\texttt{)} \\
\Pi ::= D^+
\end{array}$$

Figure 2: Syntax

Our transformations treat a higher-order, side-effect free subset of Scheme [20]. The syntax is defined in the grammar in Fig. 2. V denotes variables, K constants, O primitive operators. We assume that `lambda` and `let` expressions abstract one variable and that defined functions occur only in full applications.

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

$$\begin{array}{lcl}
\mathcal{E}[\![V]\!]\psi\rho & = & \rho[\![V]\!] \\
\mathcal{E}[\![K]\!]\psi\rho & = & \mathcal{K}[\![K]\!] \\
\mathcal{E}[\![\texttt{(if}\ E_1\ E_2\ E_3\texttt{)}]\!]\psi\rho & = & \mathit{if}\ (\mathcal{E}[\![E_1]\!]\psi\rho)\ (\mathcal{E}[\![E_2]\!]\psi\rho)\ (\mathcal{E}[\![E_3]\!]\psi\rho) \\
\mathcal{E}[\![\texttt{(}O\ E_1\ \ldots\ E_n\texttt{)}]\!]\psi\rho & = & \mathcal{O}[\![\mathrm{O}]\!](\mathcal{E}[\![E_1]\!]\psi\rho, \ldots, \mathcal{E}[\![E_n]\!]\psi\rho) \\
\mathcal{E}[\![\texttt{(}P\ E_1\ \ldots\ E_n\texttt{)}]\!]\psi\rho & = & \mathcal{E}[\![\psi(\mathrm{P})]\!][V_1 \mapsto \mathcal{E}[\![E_1]\!]\psi\rho, \ldots, V_n \mapsto \mathcal{E}[\![E_n]\!]\psi\rho] \\
\mathcal{E}[\![\texttt{(let ((}V\ E_1\texttt{))}\ E_2\texttt{)}]\!]\psi\rho & = & \mathcal{E}[\![E_2]\!]\psi\rho[V \mapsto \mathcal{E}[\![E_1]\!]\psi\rho] \\
\mathcal{E}[\![\texttt{(lambda (}V\texttt{)}\ E\texttt{)}]\!]\psi\rho & = & \lambda y.\mathcal{E}[\![E]\!]\psi\rho[V \mapsto y] \\
\mathcal{E}[\![\texttt{(}E_1\ E_2\texttt{)}]\!]\psi\rho & = & (\mathcal{E}[\![E_1]\!]\psi\rho)(\mathcal{E}[\![E_2]\!]\psi\rho)
\end{array}$$

Figure 3: A standard call-by-value interpreter

Fig. 3 shows the ancestor of which all our interpreters. The meta-language is a call-by-value lambda-calculus enriched with constants, sums, and products. The notation $\mathsf{Value*} \to \mathsf{Value}$ abbreviates the sum of $() \to \mathsf{Value}$, $\mathsf{Value} \to \mathsf{Value}$, $\mathsf{Value} \times \mathsf{Value} \to \mathsf{Value}$, 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. Our interpreters perform an equational flow analysis on the input program prior to interpretation proper. After assiging a label or *flow variable* to each expression in the program and each 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.

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 abstractions which could have generated the closure by traversing the corresponding flow class. Thanks to this binding-time improving trick [21], it is possible to keep the expression argument to the interpreter static at all times.

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

5 Bootstrapping Transformers from Interpreters

In this section, we present a range of interpreters to generate 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.

5.1 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 representation of data is crucial for effective specialization. The interpreter uses the type:

$$
\begin{array}{lcl}
\textit{Data} & ::= & \texttt{Nil} \\
& | & \texttt{Atom}\,(\mathit{Scheme\,Value}) \\
& | & \texttt{Cons}\,(\mathit{Data} \times \mathit{Data}) \\
& | & \texttt{Closure}\,(\mathit{Label} \times \mathit{Data}^*) \\
& | & \texttt{Value}\,(\mathit{Scheme\,Value})
\end{array}
$$

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 *SchemeValue*s. `Nil` and `Cons` represent 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. `Nil`, `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 E_1 E_2) with E 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. Therefore, the interpreter needs to "re-attach" (eta-expand) the constructor information to the result of a dynamic `if`:

$$
\begin{array}{l}
\texttt{(Value (}\mathit{if}\ \texttt{(get-Value}\ \mathcal{E}[\![E]\!]\psi\rho\texttt{)} \\
\qquad\qquad \texttt{(get-Value}\ \mathcal{E}[\![E_1]\!]\psi\rho\texttt{)} \\
\qquad\qquad \texttt{(get-Value}\ \mathcal{E}[\![E_2]\!]\psi\rho\texttt{)))}
\end{array}
$$

`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 dynamize all values that might be the result of a dynamic conditional.

Nontermination If a subject program has parameters that grow statically under dynamic control, the transformation cannot terminate. A prominent example are the continuation parameters of recursive functions written in continuation-passing style when control of the recursion is dynamic. The interpreter detects these situations and dynamizes the accumulating parameters.

To achieve the dynamization of closures when necessary, the interpreter dynamizes all values in the current environment when it encounters a dynamic conditional or dynamic lambda [5].

Similix inserts a *memoization point* at dynamic conditionals and dynamic lambdas. The memoization point causes Similix to generate a specialized residual function. If it encounters the same memoization point again, it re-uses the residual function, thus avoiding infinite unfolding. If—in these places—the interpreter dynamizes the environment, Similix generates such a reusable memoization point.

5.2 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. We again 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.

$$
\begin{array}{rlcl}
 & \mathsf{Value} & = & \mathsf{BaseValue} + \mathsf{Label} \times \mathsf{Var} \times \mathsf{Env} \\
\psi \in & \mathsf{ProcEnv} & = & (\mathsf{Procedure} + \mathsf{Label}) \to \mathsf{Expr}
\end{array}
$$

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

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. This approach to closure conversion has proved to be practical.

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. 5.1

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

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 and is described in Appendix B. If the interpreter avoids the above problems as described, the transformation process always terminates. An informal proof of this property is in Appendix C.

5.3 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* take the part of values. Since the specializer must handle partially static data, the dynamic components of value descriptions are references to *configuration variables* the values of which are stored in a separate environment. Value descriptions have the following form:

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

Our approach extends that of [17] by closures. Configuration variables are numbered. Consequently, the environment consists of three components: a static program variable environment mapping program variable identifiers to value descriptions, a static list of configuration variable numbers currently in use, and 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, and as operands to `car`, `cdr`, and `cons`. Tests in the conditional must take the form `(equal?` $SE\ SE$`)` or `(pair?` SE`)`. A desugaring phase before the actual interpretation converts programs into the restricted syntax. 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 [17]: When a `(let ((`$V\ E_1$`))` E_2`)` is encountered, the variable V and E_2 are put on a stack of pending `let`s, 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 so, 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 `ifs`**[1]. 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.

6 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 [32]. Mogensen reports an online partial evaluator for the λ-calculus [26]. Other online approaches that go beyond constant propagation in a first-order setting are supercompilation [30, 18], and generalized partial computation [15].

Our work extends previous work that uses the interpretive approach to achieve transformation effects. Turchin [31] shows that the interpretive approach makes transformations possible which the direct application of supercompilation to a subject program cannot perform. Glück and Jørgensen [17, 19] use the interpretive approach to generate Wadler's deforestation algorithm and a version of Turchin's supercompiler using an offline partial evaluator. 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 [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 [11] 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

[1] PT: Michael, what is this?

closure conversion is not the aim of the algorithm.

7 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.

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 nontermination.

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 ACM Symposium on Principles of Programming Languages* (1989), pp. 293–302.

[3] BONDORF, A. Automatic autoprojection of higher order recursive equations. In *European Symposium on Programming 1990* (Copenhagen, Denmark, 1990), N. D. Jones, Ed., Springer-Verlag, pp. 70–87. LNCS 432.

[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. *Sci. Comput. Programming 19*, 2 (1991), 151–195.

[6] BONDORF, A., AND JØRGENSEN, J. Efficient analysis 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 *Workshop Partial Evaluation and Semantics-Based Program Manipulation '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), vol. 300 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 236–246.

[10] CONSEL, C. A tour of Schism. In *Symp. Partial Evaluation and Semantics-Based Program Manipulation '93* (Copenhagen, Denmark, June 1993), ACM, pp. 134–154.

[11] CONSEL, C., AND DANVY, O. Static and dynamic semantics processing. In *Symposium on Principles of Programming Languages' 91* (Jan. 1991), ACM, pp. 14–24.

[12] CONSEL, C., AND KHOO, S. C. Semantics-directed generation of a prolog compiler. Tech. Rep. YALEU/DCS/RR-781, Yale University, New Haven, Connecticut, May 1990.

[13] FRIEDMAN, D. P., WAND, M., AND HAYNES, C. T. *Essentials of programming languages*. MIT Press, Cambridge, MA, 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), IEEE Computer Society Press, pp. 183–194.

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

[19] 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.

[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 *Symposium on Principles of Programming Languages '92* (Jan. 1992), ACM, ACM, 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 *ACM SIGPLAN Symp. Partial Evaluation and Semantics-Based Program Manipulation '95* (La Jolla, CA, June 1995), W. Scherlis, Ed., ACM Press, pp. 39–44.

[27] REYNOLDS, J. C. Definitional interpreters for higher-order programming. 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] TURCHIN, V. F. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems 8*, 3 (July 1986), 292–325.

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

[32] WEISE, D., CONYBEARE, R., RUF, E., AND SELIGMAN, S. Automatic online partial evaluation. In *Conf. Functional Programming Languages and Computer Architecture '91* (Cambridge, Sept. 1991), ACM, pp. 165–191.

A Flow Analysis

Our flow analysis is a straightforward adaptation of the type-based Scheme flow analysis presented in [6]. It proceeds in three steps:

flow variable generation Introduce *flow variables* ϕ_E, ϕ_V for each subexpression E and each binding occurrence of variable V in the program. For simplicity assume that all bindings bind distinct variables.

$$
\begin{array}{lll}
G(E) = & \texttt{case } E \texttt{ of} & \\
 & V & \Rightarrow \{\phi_E = \phi_V\} \\
 & K & \Rightarrow \emptyset \\
 & (\texttt{if } E_1\ E_2\ E_3) & \Rightarrow \{\phi_E = \phi_{E_2}, \phi_E = \phi_{E_3}\} \cup \bigcup G(E_i) \\
 & (O\ E_1 \ldots E_n) & \Rightarrow \{\phi_E = \phi_{E_1}, \ldots, \phi_E = \phi_{E_n}\} \cup \bigcup G(E_i) \\
 & (P\ E_1 \ldots E_n) & \Rightarrow \{\phi_E = \phi_P, \phi_{P_1} = \phi_{E_1}, \ldots, \phi_{P_n} = \phi_{E_n}\} \cup \bigcup G(E_i) \\
 & (\texttt{let } ((V\ E_1))\ E_2) & \Rightarrow \{\phi_E = \phi_{E_2}, \phi_V = \phi_{E_1}\} \cup G(E_1) \cup G(E_2) \\
 & (\texttt{lambda } (V)\ E_1) & \Rightarrow \{\langle \phi_V, \phi_{E_1}\rangle \preceq \phi_E\} \cup G(E_1) \\
 & (E_1\ E_2) & \Rightarrow \{\langle \phi_{E_2}, \phi_E\rangle \preceq \phi_{E_1}\} \cup G(E_1) \cup G(E_2) \\
\hline
\multicolumn{3}{l}{G((\texttt{define } (P\ V^*)\ E)) = \{\phi_P = \phi_E\} \cup G(E)}
\end{array}
$$

Figure 5: Generation of flow constraints [6]

constraint generation Generate flow constraints as shown in Fig. 5 which describe which expressions generate values that may flow together.

constraint simplification Apply the simplification rules in Eqn. 1 exhaustively.

The underlying type language is $\tau ::= \top \mid \tau \rightarrow \tau$ (i.e., a flow is either unknown $\top$ or a function $\tau_1 \rightarrow \tau_2$ with argument flow τ_1 and result flow τ_2). The following two kinds of constraints capture all the necessary information:

1. equality constraints of the form $\phi_1 = \phi_2$ and

2. function constraints of the form $\langle \phi_1, \phi_2 \rangle \preceq \phi_0$.

The function constraint $\langle \phi_1, \phi_2 \rangle \preceq \phi_0$ describes the flow of a function: The variable ϕ_0 describes the flow of the function itself, ϕ_1 describes the flow of the argument, and ϕ_2 the flow of the result—the function body. A solution of a set of constraints C is a mapping σ from flow variables to types—or *flow classes*—such that

1. $\forall c \in C.c \equiv \phi_1 = \phi_2 \Rightarrow \sigma(\phi_1) = \sigma(\phi_2)$ and

2. $\forall c \in C.c \equiv \langle \phi_1, \phi_2 \rangle \preceq \phi_0 \Rightarrow (\sigma(\phi_0) = \sigma(\phi_1) \rightarrow \sigma(\phi_2) \vee \sigma(\phi_0) = \sigma(\phi_1) = \sigma(\phi_2) = \top$.

In Fig. 5, ϕ_P is the flow variable of the body of procedure P and ϕ_{P_i} is the flow variable of the ith argument of P. The two rules below describe constraint simplification [6]. The rules transform a constraint set C paired with a substitution ρ mapping flow variables to flow variables.

$$
\frac{C \cup \{\phi = \phi'\}, \rho}{C[\phi \mapsto \phi'], [\phi \mapsto \phi'] \circ \rho} \qquad \frac{C \cup \{\langle \phi_1, \phi_2 \rangle \preceq \phi_0, \langle \phi'_1, \phi'_2 \rangle \preceq \phi_0\}, \rho}{C \cup \{\langle \phi_1, \phi_2 \rangle \preceq \phi_0, \phi_1 = \phi'_1, \phi_2 = \phi'_2\}, \rho} \tag{1}
$$

This rewriting system is sound and complete with respect to the set of all solutions. Furthermore, it terminates for all constraint sets. The minimal solution can be extracted from the final constraint set after exhaustive simplifcation of $(G(D), id)$ to (F, ρ) [6].

For an expression E, (F, ρ) determines the set of expressions E' the values of which flow together with E: all E' with $\rho(\phi_{E'}) = \rho(\phi_E)$. Specifically, for every application $(E\ \ E_1)$, this set determines all functions that might be applied at this point.

B Dynamization Analysis

The dynamization analysis guides the representation of closures in the interpreters to avoid non-terminating specialization as discussed in Sec. 5.2. It must ensure that

1. no static closure is ever returned through a dynamic conditional or from a dynamic closure,
2. no static closure is ever wrapped into a datastructure, and
3. no static closure contains an (indirect) reference to itself.

The interpreter should not handle any closure falling under one of these criteria statically, as non-termination may result.

To this end, let (F, ρ) be the output of the flow analysis and define an equivalence relation $\sim$ on flow variables (hence on expressions) by: $\phi \sim \phi'$ iff $\rho(\phi) = \rho(\phi')$. Define a directed graph D with the vertices flow variables and edges $\phi \rightarrow \phi'$ iff $\exists E \equiv$ `(lambda (`V'`)` E'`)` with $\phi_E \sim \phi$ and $\exists V \in FV(E)$ with $\phi_V \sim \phi'$.

Now mark flow variable ϕ as dynamic if

1. ϕ is reachable from a cycle in D,
2. $\exists E \equiv$ `(`$O\ E_1\ \ldots\ E_n$`)` $\wedge (\phi \sim \phi_{E_1} \vee \ldots \vee \phi \sim \phi_{E_n})$,
3. $\exists E \equiv$ `(if` $E_1\ E_2\ E_3$`)` $\wedge\ \phi \sim \phi_E$, and
4. $\langle \phi_1, \phi_2 \rangle \preceq \phi_0 \in F$ with $\phi_0 \sim \phi'$ for some marked ϕ' and $\phi_1 \sim \phi$ or $\phi_2 \sim \phi$.

The marking phase continues until it reaches a fixpoint.

The results the dynamization analysis are used in two places:

1. If the flow of a `lambda` is marked dynamic, the interpreter dynamizes it upon creation.
2. If the flow of the function part of an application is marked dynamic, it is treated as described in Sec. 4.

C Termination

We prove informally that the specialization of the one-level higher-order removal interpreter `int` of Sec. 5.2 with respect to arbitrary Π always terminates (even if standard evaluation of Π would not terminate). There are two reasons why a specializer `pe` might not terminate on a given interpreter `int` and program Π: infinite specialization, and a static infinite program loop.

Infinite specialization happens when an infinite number of specialized procedures is generated during specialization. Consequently, the number of different static configurations encountered at the memoization points is infinite. This never occurs, as is easily shown by inspection of the two memoization points in the interpreter: at the interpretation of conditionals and at the interpretation of applications of dynamic closures.

At a dynamic application the environment of the dynamic closure replaces the calling environment. Hence, the application of a dynamic closure depends only on the set of closures that may flow to the application—bear in mind that the free variables of a dynamic closure as well as its argument are dynamic. Since there are only finitely many such sets only finitely many versions of that code can arise.

At the interpretation of a conditional, the part of the static information that is subject to change is the expression that is currently evaluated, and the environment which is used in the evaluation. Since there are only finitely many expressions in the program, it is sufficient to ensure that only finitely many different environment descriptions arise during specialization. The only source of infinity is the constructor `Closure` as it may contain recursive occurrences of itself. This cannot occur, however, as the dynamization analysis ensures that the interpreter represents these closures dynamically.

A static infinite program loop means that the program would not terminate under normal execution. In such cases, it is questionable if any transformation is useful. However, even this case could be detected and prevented:

This case can be detected by analyzing the call-graph of Π as follows: define the *static call-graph of* Π with the set of procedures P in Π as nodes such that there is a directed edge labeled with l from P_1 to P_2 if the body of P_1 contains a call of P_2 (at $\Pi[l]$) which is not protected by a conditional or a dynamic abstraction. If the resulting static call-graph contains cycles, specializing `int` wrt. Π will not terminate. Therefore, the interpreter must choose a set of labels L such that removing the edges labeled by members of L removes all cycles from the static call-graph and that only first-order values appear at labels in L. If the latter is not possible, some closures must be dynamized in order to break all cycles. To but L into effect, the code for implementing procedure calls must first (statically) test whether the label of the current call is in L. If so, a memoization point must be inserted, otherwise evaluation proceeds as usual.

Note that the above is a conservative approximation as conditionals might be simulated using functions.

Taking together the two cases we obtain the following result: if `int` is enhanced by the static call-graph analysis described in the second case $[\![$`pe`$]\!]$ `int` Π always terminates, regardless of Π.