The Generation of a Higher-Order Online Partial Evaluator

Peter Thiemann
Fakultät für Informatik, Universität Tübingen
Sand 13, D-72072 Tübingen, Germany
E-mail: thiemann@informatik.uni-tuebingen.de

and

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

ABSTRACT

We address the problem of generating an online partial evaluator for a higher-order, functional language from an appropriate interpreter using a state-of-the-art offline partial evaluator. To ensure termination of the generated online specializer the interpreter computes a self-embedding property on closure values and data structures. This guarantees termination whenever there is no static loop in the program to be specialized. We obtain a transformer for higher-order removal and higher-order arity raising (redundancy elimination) for free, by running the online specializer on a program with completely dynamic inputs.

1. Introduction

Partial evaluation is a program specialization method based on constant propagation. In *offline* partial evaluation [21] the transformation process is guided by a *binding-time analysis* performed prior to the specialization phase. The result of the binding-time analysis is a program in which all operations are annotated as either *static* or *dynamic*. Operations annotated as static are performed at specialization time, while operations annotated as dynamic are delayed until runtime (ie residual code is generated).

Online partial evaluators [25,34] do not employ a separate binding-time analysis and transform source programs using symbolic values that can be either static values or pieces of program code. All tests whether or not an argument of an operation is a value is done on-the-fly as the computation proceeds (tag testing). If a static value occurs at any point, full use is made of that knowledge to decide conditionals and to compute other values. For example, it may be necessary to revise earlier decisions and to respecialize parts of a program. Finding a specialization strategy that always terminates and still does not lose relevant information is a difficult problem.

The problem Offline specialization methods are conceptually simpler and easier to understand than online methods due to their logical separation into two distinct stages: binding-time analysis and specialization proper. However, offline partial evaluation has its price: it is less precise because the binding-time analysis is a conservative approximation that does not take the static *values* into account and thus can cause optimizations to be missed. Furthermore, the termination of offline partial evaluation for higher-order languages is still an open problem [17].

Online partial evaluators handle binding-time information and termination issues on the fly. Both are tightly interconnected as generalization (= approximating binding-time information) is vital to obtain terminating specialization, even for simple programs. Much of the potential power of online specialization derives from the full exploitation of static values and more sophisticated descriptions of dynamic entities; cf [16,25,34]. Hence, they are usually more complex programs that may be harder to write and understand and less efficient than their offline counterpart.

This paper We try to bridge the gap between online and offline partial evaluation by *generating* an online partial evaluator from an appropriate interpreter according to the specializer projections. The contribution of this paper is twofold: (i) we generate an online specializer for a *higher-order language* using a state-of-the-art offline partial evaluator; (ii) we ensure termination of the generated specializer computing a *self-embedding property* on closure values and data structures in the interpreter. This strategy guarantees that the generated specializer terminates on static input whenever there is no static infinite loop in the source program (this is as good as the termination behavior of current offline partial evaluators).

As an interesting 'by-product' we obtain a program transformer for higher-order removal [7,2,3] and higher-order arity raising [28,24] (or redundancy elimination [29,30]), simply by running the generated online specializer on source programs with fully dynamic inputs.

The generation of a higher-order, online partial evaluator is remarkable because, for some time, it was an open problem how to *hand-write* higher-order partial evaluators. In fact, we know of only one online specializer for a significant higher-order language [34].

Specializer generation provides for a good separation of concerns: the underlying offline partial evaluator (Similix [4]) handles code generation and memoization, while the interpreter takes care of the appropriate data representation and termination issues. Similix's property not to duplicate computations is also inherited by the generated specializer. Writing the interpreter turned out to be simpler than writing a corresponding higher-order online partial evaluator from scratch. Finally, a correctness proof of the generated specializer splits naturally in two parts: the correctness of the underlying partial evaluator (shown once and for all; we take it for granted here) and the correctness of the interpreter, which is generally easier to establish than the correctness of a full specializer.

Overview We describe the specializer projections, the starting point of our work, in Section 2. The source language and the data representation of the interpreter are defined in Section 3. How termination and generalization is controlled via the interpreter is discussed in Section 4. We present the implementation in Section 5 and illustrate the results with several examples in Section 6. Related work is discussed in Section 7.

We assume that the reader is familiar with the basic concepts of partial evaluation, eg as presented by Jones and others [21], Part II.

2. Generating Specializers from Interpreters

We describe the specializer projections that specify the generation of specializers from interpreters, the starting point of our work. We adopt standard notation [21] extended with explicit binding-time classification. For any program text, `p`, written in language L we let $[\![\mathtt{p}]\!]_{\mathrm{L}}\ \mathtt{in}$ denote the application of the L-program `p` to its input `in`. We use a `typewriter` font for programs and their input and output.

Preliminaries Suppose `p` is a source program, $\mathtt{in}_1$ its static input and $\mathtt{in}_2$ its dynamic input. Then the computation in one stage is described by

$$\mathtt{out} = [\![\mathtt{p}]\!]_{\mathrm{L}}\ \mathtt{in}_1\ \mathtt{in}_2$$

Computation in two stages using an L→L-*specializer* `spec` written in L is described by the *mix equation* [21]

$$[\![\mathtt{p}]\!]_{\mathrm{L}}\ \mathtt{in}_1\ \mathtt{in}_2 = [\![\ [\![\mathtt{spec}]\!]_{\mathrm{L}}\ \mathtt{p}\ \text{`}\mathtt{SD}\text{'}\ [\mathtt{in}_1]\,]\!]_{\mathrm{L}}\ \mathtt{in}_2$$

where the binding-time classification `'SD'` indicates the binding times of `p`'s input (the first argument is static `'S'`, the second is dynamic `'D'`) and the list `[ ]` contains `p`'s input static input; multi-language specialization is treated by the second author [13].

Let `int` be an S-*interpreter* written in L. For notational convenience we assume that the interpreter `int` accepts three inputs: an S-program `src` and `src`'s input distributed over two arguments $\mathtt{dat}_1$ and $\mathtt{dat}_2$. Then the interpretation of S-programs can be described equationally by

$$[\![\mathtt{int}]\!]_{\mathrm{L}}\ \mathtt{src}\ \mathtt{dat}_1\ \mathtt{dat}_2 = [\![\mathtt{src}]\!]_{\mathrm{S}}\ \mathtt{dat}_1\ \mathtt{dat}_2$$

Futamura Projections The *Futamura projections* [10] assert that compilation can be achieved and that compilers and compiler generators can be produced by using an interpreter and a self-applicable program specializer. For example, the first Futamura projection states how to compile an S-program `src` by specializing the S-interpreter `int`. The compiled version of the S-program `src` is the L-program `target`.

$$\mathtt{target} = [\![\mathtt{spec}]\!]_{\mathrm{L}}\ \mathtt{int}\ \text{`}\mathtt{SDD}\text{'}\ [\mathtt{src}]$$

where the binding-time classification `'SDD'` indicates the binding times of `int`'s input. It is easy to verify that `target` is a translated version of `src` by combining the above equations.

$$[\![\mathtt{target}]\!]_{\mathrm{L}}\ \mathtt{dat}_1\ \mathtt{dat}_2 = [\![\mathtt{src}]\!]_{\mathrm{S}}\ \mathtt{dat}_1\ \mathtt{dat}_2$$

Specializer Projections The *specializer projections* [13] assert that specialization can be achieved and that specializers can be produced by using an interpreter and a self-applicable program specializer. For example, the first specializer projection states how to specialize an S-program `src` by specializing an S-interpreter with an L→L-specializer. We assume that $\mathtt{dat}_1$ is `src`'s static input and $\mathtt{dat}_2$ its dynamic input. The specialized version of the S-program `src` is the L-program `resid`.

$$\mathtt{resid} = [\![\mathtt{spec}]\!]_{\mathrm{L}}\ \mathtt{int}\ \text{‘}\mathtt{SSD}\text{’}\ [\mathtt{src}\ \mathtt{dat}_1]$$

where the binding-time classification `'SSD'` indicates the binding times of `int`'s input. It is easy to verify that `resid` is a specialized version of `src` by combining the above equations.

$$[\![\mathtt{resid}]\!]_{\mathrm{L}}\ \mathtt{dat}_2 = [\![\mathtt{src}]\!]_{\mathrm{S}}\ \mathtt{dat}_1\ \mathtt{dat}_2$$

Similar to the *2nd Futamura projection*, which defines the generation of an S→L-compiler, the *2nd specializer projection* defines the generation of an S→L-specializer from an S-interpreter. Using the special case where S⊆L, called ‘self-generation’ [13], we can generate a specializer with (a subset) of Scheme as source and target language, but properties drastically different from the original specializer.

Note that the specializer projections allow the generation of offline specializers, too. Let `bta` be a binding-time analysis that returns annotated programs written in the language S_{ann} and let `intann` be an interpreter for S_{ann}. Then the interpretation of an S-program can be performed into two steps

$$[\![\mathtt{intann}]\!]_{\mathrm{L}}\ ([\![\mathtt{bta}]\!]_{\mathrm{L}}\ \mathtt{src}\ \text{‘}\mathtt{SSD}\text{’})\ \mathtt{dat}_1\ \mathtt{dat}_2 = [\![\mathtt{src}]\!]_{\mathrm{S}}\ \mathtt{dat}_1\ \mathtt{dat}_2$$

where `'SSD'` is some binding-time classification. Consequently, the specializer projections describe the generation of an S_{ann}→L-specializer phase from the S_{ann}-interpreter `intann`.

3. The Interpreter and its Data Representation

3.1. The Source Language

The source language of the interpreter is a higher-order, side-effect free subset of Scheme [20]. In the EBNF grammar in Fig. 1, V denotes variables, K constants, O primitive operators like `+`, `-`, `cons`, ..., P denotes defined procedures, E expressions, D procedure definitions, and finally Π programs. For simplicity of the implementation we assume that lambda abstractions have exactly one parameter and that not all primitive operators of Scheme are available.

3.2. Data Representation and Partially Static Structures

Choosing the data representation is a key issue in the design of the interpreter. Its careful choice allows us to control memoization and thus the termination characteristics of the generated partial evaluator. We can enforce generalization by changing the representation of data in the interpreter.

Table 1. Syntax

$$
\begin{array}{rcl}
E & ::= & V \mid K \mid \texttt{(if } E_1\ E_2\ E_3\texttt{)} \mid \texttt{(}O\ E^*\texttt{)} \mid \texttt{(}P\ E^*\texttt{)} \mid \\
 & & \texttt{(let ((}V\ E_1\texttt{))}\ E_2\texttt{)} \mid \texttt{(lambda (}V\texttt{)}\ E\texttt{)} \mid \texttt{(}E_1\ E_2\texttt{)} \\
D & ::= & \texttt{(define (}P\ V^*\texttt{)}\ E\texttt{)} \\
\Pi & ::= & D^+
\end{array}
$$

All data manipulated by the interpreter is represented as elements of an algebraic type

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

The arguments of `Atom` are first-order, atomic *SchemeValue*s. Data structures are built using `Nil` and `Cons`. (Partially) Static closures are represented by the constructor `Closure` applied to the unique label of the corresponding lambda abstraction and the list of the values of the free variables. The argument of the `Value` constructor is an arbitrary Scheme value, atomic or structured, first-order or higher-order.

The crucial point is the distinction between `Value` on one hand and `Atom`, `Nil`, `Cons`, and `Closure` on the other hand in the interpreter. The binding-time of the argument of the former will be dynamic, while the binding-time of the arguments of the latter group will be static or partially static. Hence, we call data represented by `Value` *dynamically represented* and call all other data representations static. It is easy to write a static conversion procedure (executable at partial evaluation time) that transforms data from the static representation into the dynamic representation. The conversion from the static into the dynamic representation can be used to dynamize ('lift') static data. This allows us to lift higher-order and structured data, whereas standard offline partial evaluators only lift first-order, atomic data while higher-order and structured data are created dynamic right away if they are ever used in a dynamic context.

3.3. Technical Remarks

Our task is considerably simplified by exploiting *partially static structures*, a feature offered by most state-of-the-art offline specializers (eg Similix [4], Schism [8]). This saves us from writing a two-level interpreter that internally maintains two separate environments in order to keep static representations separate from dynamic representations, an approach employed by Glück and Jørgensen [14,15] to generate specializers using partial evaluators which do not provide partially static structures such as Mix and Unmix.

Technically, we are going to generate the partial evaluator without actually writing an interpreter with three arguments: given an ordinary (one level) interpreter `int`, an

S-program `src`, and the static part of its input `dat1` we first transform `src` as follows. If `(define (main` $\mathtt{d}_1$ ... $\mathtt{d}_n$`) ...)` is the main procedure of `src` and $s_1, \ldots, s_k$ comprises the static input $\mathtt{inp}_S$ we add a new main procedure `main0`, defined by:

$$\texttt{(define (main0 } \mathtt{d}_{k+1} \ \ldots \ \mathtt{d}_n\texttt{)}$$
$$\texttt{(main } s'_1 \ \ldots \ s'_k \ \mathtt{d}_{k+1} \ \ldots \ \mathtt{d}_n\texttt{))}$$

In the body of `main0`, s'_i is the *activated* version of the static data s_i: s'_i is an expression which produces s_i when executed, quite similar to backquoting s_i. When we now apply `spec` to the interpreter `int` and the transformed program we obtain a specialized program.

4. Generalization and Termination Issues

Looking at the data representation introduced in the previous section through the eyes of the specializer, we see that computations involving `Value`s generate residual code, while all computations involving other data are performed at partial evaluation time. Therefore, if the interpreter decides to defer some computation it just changes its representation from static to dynamic, ie to `Value`.

Notice that we are only ever specializing the interpreter using the offline partial evaluator. Therefore we only have to worry that this specialization terminates for as many different static inputs as possible. This is considerably simpler than having to devise a termination strategy for arbitrary programs.

4.1. Ensuring Termination with Birthplaces

A *birthplace* uniquely identifies a program point where data is created. This is a common method in non-standard semantics, which are subsequently used for abstract interpretation [1,18,27]. By augmenting all static data representations with birthplaces, we make the transition from a standard interpreter to a non-standard one. The birthplaces will be used to make the specialization of the interpreter terminate.

The closure representation already carries its birthplace in form of the unique label identifying the corresponding lambda abstraction. Additionally we need to extend the representation of `Cons` cells with a new field that registers its birthplace (some unique description of the program point where the `Cons` cell is constructed). The information in this field is obviously static.

Constants of finite enumeration types are not critical for termination and can safely be ignored, ie they need not be tagged with a birthplace. For numbers we confine ourselves to natural numbers and tag each of them with a set of birthplaces. Whenever an addition of two numbers is performed their sets of birthplaces are unioned and the location of the addition operation is included. Again, this is a static computation.

4.1.1. Where to Generalize?

In order to guarantee termination in the absence of static loops we need to appropriately generalize all entries in the environment whenever the underlying specializer creates a memoization point (performs caching of residual functions). If the interpreter just changes the representation from static to dynamic, the memoization mechanism of the underlying partial evaluator will do the rest. A well-known, practical method which delivers fairly good termination behavior is to insert memoization points at *dynamic conditionals* [5], ie in the interpreter at the *interpretation* of the conditional. This strategy is based on the following observation: any non-trivial loop in a program contains at least one conditional for deciding whether to stop or to continue looping (loops without such a conditional never terminate and we don't take any responsibility). If all these conditionals are static partial evaluation will terminate on this loop whenever ordinary evaluation does. If at least one of the conditions is dynamic the specializer can refrain from looping by memoizing previous configurations.

4.1.2. Self-Embedding of Closures and Data Structures

Now the question remains what do we need to generalize in a piece of data. Our strategy relies on the fact that there is only a finite number of lambda abstractions, cons operations, and additions in a given program.[a]

Closures. Static closures can only give rise to an infinitely growing data structure (in the interpreter) if they are *self-embedding*, ie a closure with label X appears in the value of a free variable of another closure with the same label X. This condition can be tested by a simple traversal of the values of the free variables on creation of the closure. If the label of the current lambda occurs in the value of a free variable, the representation of that particular part is changed to dynamic. Thus, a generalization is achieved.

Consider the following example:

```
(define (length-c xs cont)
  (if (null? xs)
      (apply cont 0)
      (call length-c (cdr xs)
            (lambda (z) (apply cont (+ z 1))))))
```

In the example the closure for `(lambda (z) ...)` is self-embedding as its only free variable `cont` may be bound to a closure of the same lambda abstraction.[b]

[a] We assume that input values contain unique birthplaces.

[b] Notice that our online specializer terminates on `length-c` which is problematic with most current offline specializers.

Data structures. Data structures are handled in the same way as closures. The only change is that birthplaces take the place of closure labels.

Natural numbers. Our treatment of natural numbers is inspired by the treatment they would receive if they were represented by the recursive data type

$$Nat = \texttt{Zero} \mid \texttt{Succ}\ Nat$$

Each application of the `Succ` constructor would carry its own unique birthplace. As there are no more `Succ` constructors we approximate the set of all birthplaces of the `Succ` constructors in a number by the set of places where an addition is performed on the number. Whenever an addition is attempted whose place is already in the set of addition places of one of the arguments, the entire operation is deferred and the value is converted to the dynamic representation.

It may be possible to achieve more precise results by having three representations for numbers: completely static (`Atom` n), completely dynamic (`Value` n), and additionally partially static numbers where it is known that some operations can be performed on the static part. The most interesting operations would obviously be inequality tests, because some of them could yield a completely static result.

4.2. On- or Offline Termination Analysis

There are two ways to perform the generalization. The analysis described above is online: run the interpreter up to the next dynamic conditional, scan the environment for statically self-embedding values and change their representation from the static to the dynamic representation.

The other solution is an offline analysis: determine which closures and data structures are possibly self-embedding and create them in the dynamic representation right away. Such an approach is described by the present authors [31] where the necessary information is extracted from a data flow analysis which is similar to an analysis used in the partial evaluator Similix. Exactly the same can be done for data structures and hence also for natural numbers.

While the online analysis potentially leads to deeper specialization some of its work can be redundant. For example, it is possible that the same static data has its representation changed to dynamic several times (this results in code duplication in the residual program). Using an offline analysis, data never changes its representation, because it is already created in the representation in which it is going to be processed. Therefore, the offline approach is much more conservative in duplicating code.

When changes in the data representation are guided by a program annotation (as suggested in Sec. 2) we could in fact generate an offline partial evaluator with arbitrary characteristics.

4.3. Type System vs. Data Representation

Our data representation may be seen as a preimage of the type system of a (sophisticated) online partial evaluator [25] under the offline specializer. The corresponding type system distinguishes closures and data structures from first-order atomic values. The type $\top$ corresponds to the `Value` summand in the data representation and computations on it are deferred, accordingly. Subtypes like `Bool` or `Symbol` are implicit in the `Atom` summand, but they are only kept separate by the underlying Scheme system. Therefore, we do not have something corresponding to $\top_{\texttt{Bool}}$ denoting some value which is certainly boolean, but it is not known whether it is `#t` or `#f`. It would, however, make sense to introduce more structure in the type *Data* in a Scheme implementation where untagged primitives are available. This way, part of the tag checking work could be left to the partial evaluator, as well. The subtypes for closures `Closure` and data structures `Nil` and `Cons` are constructed in the same way as in an actual online partial evaluator.

5. Implementation

All that is described in the present work is actually implemented in Scheme using the Scheme48 system [22] and the Similix partial evaluator, version 5.1 [4]. We have generated program transformers and specializers which achieve higher-order removal [7,6,2], arity raising [28,24], redundancy elimination [29,30], and other interesting effects (see below for a sample of representative examples).

One of the main advantages of using Similix is, that it guarantees that no computation in a source program is duplicated. The generated specializer inherits this property as long as there is no computation duplication in the interpreter from which the specializer was generated. Note that we get this property for free!

In our implementation the representation of closures, data structures, and atomic values is handled on the fly, ie online. It is interesting to note that the interpreter is written entirely in first-order style, except from two obvious places:

1. when the representation of a closure is changed from static to dynamic the dynamic function is generated using a lambda-expression,

2. when a dynamically represented function is applied to its argument, a function application is performed in the interpreter.

It is also possible to eliminate these two places. Doing so exhibits the closure representation chosen in the interpreter to the residual (transformed) program and thus would perform a closure conversion as has been shown by the present authors [31].

Residual programs may contain redundant specializations because the representation of data in the interpreter includes birthplaces, ie data that is extensionally

Table 2. A fragment of the interpreter: interpretation of the conditional

```
eval env (If e1 e2 e3) =
  let b1 = eval env e1
  in
    case b1 of
      Atom  b => if b then eval env e2
                      else eval env e3
    | Value x =>
      let v0   = freevars e1 ∪ freevars e2
          env' = restrict_and_dynamize env v0
      in
        Value (
          mk_memoization_point (
           if x then getValue (dynamize (eval env' e1))
                else getValue (dynamize (eval env' e2))))
      end
    | _ => eval env e2
  end
```

The code is a simplified transliteration of the Scheme code to SML (for readability). Notice that `b` is static and hence the first conditional, `x` is dynamic (and hence the second conditional), `restrict_and_dynamize` performs generalization as discussed in Sec. 4, `getValue` projects out of the `Value` summand of *Data*, `dynamize` transforms its *Data* argument into the dynamic representation, `mk_memoization_point` sets a memoization point, the default case of the match reflects the fact that everything but `#f` is 'true' in Scheme.

the same, but was created at different places, is considered as different. A more sophisticated comparison (or labeling) strategy could be used to collapse intensionally different representations.

A higher-order online specializer has been generated from the interpreter (Fig. 2) using the compiler generator supplied with Similix. This took 155.66 seconds and the size of the generated specializer is 11890 cons-cells. Runtimes of the specializer are listed with the respective programs. Note that Scheme48 is an interpreter-based implementation. A compiled Scheme implementation can be faster by a factor of 20.

6. Applications

The generated specializer is too large to be shown here, but we discuss and illustrate some of its applications.

6.1. Higher-Order Specialization and Removal

Higher-order specialization is often tedious with offline specializers. As higher-order functions often abstract common recursion patterns, they are often used at several different binding times. If the specializer has a monovariant binding-time analysis there is no option but to either duplicate the definition or to rewrite the program to first-order style. On the other hand, if a polyvariant binding-time analysis is present the specializer may easily run in termination problems (as pointed out in Sec. 2).

6.1.1. Examples for Higher-Order Specialization

The first example uses the higher-order function `map` with different binding times: once with a static list, and once with a dynamic one.

```
(define (f0 z x)
  (if z (call f1 x)
        (call f2 x)))
(define (f1 x)
  (call map (lambda (z) (+ z x)) '(1 2 3)))
(define (f2 x)
  (call map (lambda (z) (+ z z)) x))
(define (map f x)
  (if (null? x) '()
      (cons (apply f (car x))
                 (call map f (cdr x)))))
```

With `z` and `x` dynamic we obtain the following residual code in 8.40 seconds:

```
(define (s1-2int-0 xd*_0)
 (define (s1-eval-0-2 g-b-16_0 g-a*-19_1)
   (if g-b-16_0
       '()
       (let* ((g_2 (car g-a*-19_1)) (g_4 (cdr g-a*-19_1)))
         (cons (+ g_2 g_2) (s1-eval-0-2 (null? g_4) g_4)))))
 (let ((g_3 (car (cdr xd*_0))))
   (if (car xd*_0)
       (list (+ 1 g_3) (+ 2 g_3) (+ 3 g_3))
       (s1-eval-0-2 (null? g_3) g_3))))
```

The call to `f1` is evaluated up to the unknown value of `x`. The call to `f2` generates a call to function `s1-eval-0-2` which is `map` specialized wrt. `(lambda (z) (+ z z))`. Notice that the binding times of the two occurrences of `map` are different: dynamic and static list. Using an offline approach with monovariant binding-times the call to `f1` would only have resulted in another specialized version of `map`.

Further specialization with respect to `z = #t` yields (in 5.81 seconds):

```
(define (s1-2int-0 xd*_0)
 (let ((g_1 (car xd*_0)))
   (list (+ 1 g_1) (+ 2 g_1) (+ 3 g_1))))
```

6.1.2. Higher-order removal

The following program is taken from a paper on higher-order removal [7]. It is a version of `map` which accepts two function and applies them in turn to successive list elements.

```
(define (m_alt xs)
  (call altmap (lambda (x) (* x x)) (lambda (x) (* x (* x x))) xs))
(define (altmap f g xs)
  (if (null? xs) '()
      (cons (apply f (car xs))
                 (call altmap g f (cdr xs)))))
```

Running the specializer on it with unknown input `xs` takes 5.46 seconds and yields:

```
(define (s1-2int-0 xd*_0)
 (define (s1-eval-0-1 g-b-16_0 g-a*-19_1)
   (if g-b-16_0
       '()
       (let* ((g_2 (car g-a*-19_1)) (g_4 (cdr g-a*-19_1)))
         (cons (* g_2 g_2)
               (if (null? g_4)
                   '()
                   (let* ((g_6 (car g_4)) (g_9 (cdr g_4)))
                     (cons (* g_6 (* g_6 g_6))
                           (s1-eval-0-1 (null? g_9) g_9))))))))
 (let ((g_1 (car xd*_0)))
   (s1-eval-0-1 (null? g_1) g_1)))
```

The body of `s1-eval-0-1` contains two variants of `altmap` one specialized wrt. `(lambda (x) (* x x))` and the other wrt. `(lambda (x) (* x (* x x)))`.

6.1.3. Limitations

Premature generalization can be provoced with an example due to Sestoft [26] to defeat his "inductive variable" criterion for unfoldable function calls.

```
(define (main x)
  (if (> x 0)
    (call main (- (+ x 1) 2))
    0))
```

Even for static `x = 5` we obtain:

```
(define (s1-2int-0 xd*_0)
  (define (s1-eval-0-1 g-b-16_0 g-a*-19_1)
    (if g-b-16_0
        (let ((g_3 (- (+ g-a*-19_1 1) 2)))
          (s1-eval-0-1 (> g_3 0) g_3))
        0))
  (s1-eval-0-1 #t 3))
```

This is far from the optimal result `(define (main-0) 0)`.

6.2. Applications of Specializer Generation

This contribution provides an example for some of the applications of specializer generation discussed theoretically by the second author [13]. We will now discuss them briefly.

- *Bootstrapping*: our interpreter is (almost) exclusively written in a first-order subset of Scheme, while the generated specializer accepts a higher-order subset of Scheme as input. In other words, a higher-order specializer has been bootstrapped from a first-order specializer via an appropriate interpreter.
- "*Bricks instead of tricks*": we changed the working principle of a specializer from offline to online by encapsulating this feature in an interpreter; instead of writing a new specializer from scratch.
- *Generic specializers*: it is easy to imagine that similar interpreters can be written for other higher-order languages from which we generate new specializers. In other words, Similix then plays the role of a generic specializer for a variety of new specializers.

7. Related Work

The interpretive approach has been proposed by Turchin [33] in the context of supercompilation. The generation of optimizing specializers by specialization of interpreters has been suggested by the second author [13]. It is a surprising fact that, given an appropriate interpreter, the interpretive method may drastically change the properties of the overall transformation, eg online transformers for supercompilation and deforestation have been generated [14,15]. The present work explores the interpretive approach by controlling the termination behavior of a transformation process via an interpreter.

The first step towards transformers for higher-order languages is taken by the present authors [31] where a specializer for a higher-order language is generated by specializing an interpreter written in a first-order subset of Scheme using a specializer for a first-order language. As a by-product, a closure conversion algorithm has been derived which also performs higher-order removal to some extent.

The specializer presented in the current work is also able to perform higher-order removal. It does so in a more effective way than the algorithm generated in previous work [31] as it does not confuse functions that are put into data structures by employing a proper data representation. Data representation is not just a technical issue, but also essential for the self-application of online partial evaluators, as discussed by the second author [12].

Many of the effects of arity raising [24,28] can be achieved using our specializer. Higher-order redundancy elimination as described by the first author [30] is also possible in principle, but the current specializer does not yet propagate sufficient information to achieve all effects described in that work.

The only online partial evaluator for a realistic higher-order, functional language we know of is Fuse, described in depth by Ruf [25] and Weise and others [34]. Recently Mogensen reported an online partial evaluator for the λ-calculus [23]. Other online approaches that go beyond constant propagation are supercompilation [32] (see also [16]), a unification-based transformation technique, and generalized partial computation [11], a method that assumes the power of a theorem prover.

Acknowledgments Special thanks are due to Dirk Dussart and Jesper Jørgensen for many interesting discussions.

8. References

1. S. Abramsky and C. Hankin. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.
2. J. M. Bell. An implementation of Reynold's defunctionalization method for a modern functional language. Master's thesis, Oregon Graduate Institute of Science and Technology, Portland, Oregon, USA, Nov. 1993.
3. J. M. Bell and J. Hook. Defunctionalization of typed programs. Feb. 1994.
4. A. Bondorf. *Similix 5.0 Manual.* DIKU, University of Copenhagen, May 1993.
5. A. Bondorf and O. Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Programming*, 19(2):151–195, 1991.
6. W.-N. Chin. Fully lazy higher-order removal. In C. Consel, editor, *Workshop Partial Evaluation and Semantics-Based Program Manipulation '92*, pages 38–47, San Francisco, CA, June 1992. Yale University. Report YALEU/DCS/RR-909.
7. W.-N. Chin and J. Darlington. A higher-order removal method. Submitted for publication, Sept. 1994.
8. C. Consel. A tour of Schism. In D. Schmidt, editor, *Symp. Partial Evaluation and Semantics-Based Program Manipulation '93*, pages 134–154, Copenhagen, Denmark, June 1993. ACM.
9. G. Filé, editor. *3rd International Workshop on Static Analysis*, Padova, Italia, Sept. 1993. Springer-Verlag. LNCS 724.
10. Y. Futamura. Partial evaluation of computation process — an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
11. Y. Futamura, K. Nogi, and A. Takano. Essence of generalized partial computation. *Theoretical Comput. Sci.*, 90(1):61–79, 1991.
12. R. Glück. Towards multiple self-application. In *Proc. Partial Evaluation and Semantics-Based Program Manipulation '91*, pages 309–320, New Haven, June 1991. ACM. SIGPLAN Notices 26(9).
13. R. Glück. On the generation of specializers. *Journal of Functional Programming*, 4(4):499–514, Oct. 1994.
14. R. Glück and J. Jørgensen. Generating optimizing specializers. In *IEEE International Conference on Computer Languages 1994*, pages 183–194, Toulouse, France, 1994. IEEE Computer Society Press.
15. R. Glück and J. Jørgensen. Generating transformers for deforestation and supercompilation. In B. Le Charlier, editor, *Proc. International Static Analysis Symposium, SAS'94*, pages 432–448, Leuven, Belgium, Apr. 1994. Springer-Verlag. LNCS 864.
16. R. Glück and A. V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In Filé [9], pages 112–123. LNCS 724.

17. C. K. Holst. Finiteness analysis. In Hughes [19], pages 473–495. LNCS 523.
18. P. Hudak. *A Semantics Model of Reference Counting and its Abstraction*, chapter 3, pages 45–62. In [1], 1987.
19. J. Hughes, editor. *Functional Programming Languages and Computer Architecture*, Cambridge, MA, 1991. Springer-Verlag. LNCS 523.
20. IEEE. *Standard for the Scheme programming language.* Institute of Electrical and Electronic Engineers, Inc., 1991. Tech. Rep. 1178-1990.
21. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.
22. R. A. Kelsey and J. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1994.
23. T. Æ. Mogensen. Self-applicable online partial evaluation of pure lambda calculus. In W. Scherlis, editor, *ACM SIGPLAN Symp. Partial Evaluation and Semantics-Based Program Manipulation '95*, pages 39–44, La Jolla, CA, June 1995. ACM Press.
24. S. A. Romanenko. Arity raiser and its use in program specialization. In N. D. Jones, editor, *Proc. 3rd European Symposium on Programming 1990*, pages 341–360, Copenhagen, Denmark, 1990. Springer-Verlag. LNCS 432.
25. E. Ruf. *Topics in Online Partial Evaluation.* PhD thesis, Stanford University, Stanford, CA 94305-4055, Mar. 1993. Technical report CSL-TR-93-563.
26. P. Sestoft. Automatic call unfolding in a partial evaluator. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 485–506, Amsterdam, 1988. North-Holland.
27. O. Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, May 1991. Also technical report CMU-CS-91-145.
28. B. Steensgaard and M. Marquard. Parameter splitting in a higher-order functional language. DIKU Student Project 90-7-1, DIKU, University of Copenhagen, 1990 1990.
29. P. Thiemann. Avoiding repeated tests in pattern matching. In Filé [9], pages 141–152. LNCS 724.
30. P. Thiemann. Higher-order redundancy elimination. In P. Sestoft and H. Søndergaard, editors, *Workshop Partial Evaluation and Semantics-Based Program Manipulation '94*, pages 73–84, Orlando, Fla., June 1994. ACM.
31. P. Thiemann, R. Glück, and M. Sperber. Closure conversion and higher-order removal by partial evaluation. Unpublished manuscript, Apr. 1995.
32. V. F. Turchin. The concept of a supercompiler. *ACM Trans. Prog. Lang. Syst.*, 8(3):292–325, July 1986.
33. V. F. Turchin. Program tranformation with metasystem transitions. *Journal of Functional Programming*, 3(3):283–313, July 1993.
34. D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In Hughes [19], pages 165–191. LNCS 523.