How To Have Your Cake and Eat It, Too: Self-Applicable Online Partial Evaluation

Michael Sperber

Wilhelm-Schickard-Institut für Informatik
Universität Tübingen
Sand 13, D-72076 Tübingen, Germany
sperber@informatik.uni-tuebingen.de
phone +49 (7071) 29-5467, fax +49 (7071) 29-5958

Abstract. We propose a hybrid approach to partial evaluation to achieve self-application of realistic online partial evaluators. Whereas the offline approach to partial evaluation leads to efficient specializers and self-application, online partial evaluators perform better specialization at the price of efficiency. Moreover, no online partial evaluator for a realistic language has been successfully self-applied. We present a binding-time analysis for an online partial evaluator for a higher-order subset of Scheme. The analysis distinguishes between *static*, *dynamic*, and *unknown* binding times. Thus, it makes some reduce/residualize decisions offline while leaving others to the specializer. The analysis does not introduce unnecessary generalizations. We have implemented the binding-time analysis and an online specializer to go with it. After some standard binding-time improvements, our partial evaluator successfully self-applies. Moreover, it is amenable to effective specialization with an *offline* partial evaluator.

Keywords: online partial evaluation, offline partial evaluation, self-application, compiler generation, binding-time analysis

1 Introduction

Partial evaluation is a program specialization technique based on aggressive constant propagation. During specialization, a partial evaluator reduces expressions with known values while residualizing those whose values are not known at specialization time. Partial evaluators can make the residualize/reduce decision in two different ways: *Offline* partial evaluators make the decision before specialization itself by performing an "offline" *binding-time analysis* which annotates each expression in the program as either reducible or subject to residualization. *Online* partial evaluators, on the other hand, make the decision during specialization, based on the actual values propagated.

Both techniques have respective advantages: The specialization phase of an offline partial evaluator merely follows blindly the annotations made by the binding-time analysis. It is therefore comparatively simple and efficient. Moreover, it makes offline partial evaluators amenable to *self-application* which makes it possible to generate stand-alone compilers and compiler generators. However, the binding-time analysis can provide only an approximation of the true binding-time behavior of the subject program, which is often crude and makes for poor precision during specialization. The online approach, on the other hand, can generally perform better specialization at the cost of specialization efficiency. Also, the complex specialization phase and the lack of offline binding-time information make self-application of a straightforward online partial evaluator non-practical.

We show how to unify the advantages of both approaches by using an online specialization phase in conjunction with an offline binding-time analysis. As opposed to binding-time analyses in offline systems, our analysis admits when it cannot precisely determine a binding time; hence it distinguishes between binding time values *static*, *dynamic*, and *unknown*. The analysis is able to determine that substantial parts of the specialization phase are indeed static which makes self-application possible.

To summarize:

- We have specified a binding-time type system for a substantial higher-order functional subset of Scheme which allows for unknown binding times.
- We have implemented an efficient binding-time analysis based on a union-find-based constraint-solving algorithm along the lines of Henglein's work [12].
- We have implemented a specializer which works on binding-time-annotated subject programs; it handles higher-order functions, partially static data structures, and performs arity raising.
- We have performed single and double self-application of the specializer, and generated stand-alone compilers.
- We have generated stand-alone compilers by applying offline partial evaluation to our specializer.

Overview The next section gives a brief account of online partial evaluation. In Sec. 3, we explain why self-application is harder to achieve with the online approach, and how to correct the problem by using a binding-time analysis. Subsequently, Sec. 4 briefly explains the type system underlying our binding-time analysis. Section 5 explains how to use its results in a specializer and Sec. 6 shows how to improve the binding times of the specializer to achieve self-application. Some preliminary experimental results are contained in Sec. 7. Section 8 contains on overview of related work.

2 Online Partial Evaluation

Online partial evaluators propagate *symbolic values* through a program [30, 19]. A symbolic value is a description of a set of runtime values. and represents the specialization-time approximations of runtime values. Typically, it will either contain an actual value already computed at specialization time, or some code to compute a value not known until runtime, possibly with additional type information. Thus, symbolic values may have the following structure for a higher-order language with partially static data structures:

$$\begin{aligned} SVal ::= &\ Top(Code) \\ &\mid Scalar(Val) \\ &\mid PS(CTor, SVal^*) \\ &\mid Closure(Code, SVal^*) \end{aligned}$$

In this specification, *Code* is a representation for output code, *Val* is for first-order Scheme values, *CTor* is for constructors of partially static data (such as `cons`); a *Closure* object needs to contain code for its body, and the symbolic values of its free variables.

$$\begin{array}{l}
spec[\![\texttt{(if } E_0\ E_1\ E_2\texttt{)}]\!] = \\
\quad \textbf{let } sval_0 = spec[\![E_0]\!] \\
\quad \textbf{in} \\
\qquad \textbf{case } sval_0 \textbf{ of} \\
\qquad\quad Scalar(v_0) \rightarrow \textbf{if } v_0 \textbf{ then } spec[\![E_1]\!] \textbf{ else } spec[\![E_2]\!] \\
\qquad\quad Top(c_0) \quad\ \rightarrow \textbf{let } c_1 = \mathcal{C}(spec[\![E_1]\!]) \\
\qquad\qquad\qquad\qquad\quad\ \ c_2 = \mathcal{C}(spec[\![E_2]\!]) \\
\qquad\qquad\qquad\qquad \textbf{in } \ Top(\texttt{(if } c_0\ c_1\ c_2\texttt{)})
\end{array}$$

Fig. 1. Code fragment from an online partial evaluator for handing `if`

The specialization phase itself is a special interpreter which executes a program on symbolic values. Hence, its environment also contains symbolic values. For each operation, the specializer checks if it can execute the operation at specialization time, and, based on that decision, creates either residual code via *Top* or reduces the expression, yielding a *Scalar*, a *PS*, or a *Closure* object. Figure 1 contains an example for handling `if` in an online partial evaluator. The $\mathcal{C}$ function used there generates code from a symbolic value. The example shows that an online partial evaluator, in contrast to its offline counterpart, need not generalize `if`s with branches that have different binding times.

Since an online partial evaluator can make reduce/residualize decisions at runtime, its specialization is necessarily more accurate than that of offline counterparts. Specifically, online partial evaluators are trivially binding-time polyvariant, and therefore more likely to perform useful specialization on programs not optimized for effective partial evaluation. Realistic programs, when subjected to a monovariant offline analysis, get most binding-times coerced to "dynamic" by binding-time-polyvariant calls [19, 23].

3 Self-Application and Compiler Generation

Researchers in online partial evaluation have traditionally favored specialization accuracy over the possibility of self-application [30]. The main motivation for self-

application is the generation of stand-alone compilers from interpreters [4, 15, 19]. However, self-application of realistic partial evaluators so far has only been possible for the offline case—on the other hand, actually obtaining *optimizing* compilers requires some online specialization [19]. Consequently, it is certainly desirable to achieve at least compiler generation using the online approach.

The main problem in achieving compiler generation with the online approach is the lack of binding-time information in the specializer being specialized. In the offline case, compiler generation works as follows:

$$\mathtt{comp} = [\![\mathtt{pe}]\!](\mathtt{pe}^{sd}, \mathtt{int}^{sd})$$

In the example, $\mathtt{pe}^{sd}$ and $\mathtt{int}^{sd}$ are binding-time-annotated sources for the partial evaluator and the interpreter, respectively. Now, in the online case, there are no binding-time annotations. Hence, it is not known to the specializer if the interpreter is to be specialized with respect to a static program and dynamic input (the intended effect), or with respect to a static input and a dynamic program—a not very useful feature. Thus, the resulting compiler suffers from excess generality [14], and exhibits little or no performance advantage, and usually significant bloat of the resulting program. Therefore, the lack of binding-time information is at the heart of the problem. The question is therefore how to introduce binding-time information without compromising the precision of the specialization process.

Standard offline binding-time analyses are often indeed crude approximations of the true binding-time behavior of a given program. However, it is only crude in the sense that binding times which the analysis cannot prove to be static are coerced to dynamic. The information about static values is precise. Since we are really interested in having our specializer recognize static operations, and just perform full online specialization on the rest, an analysis capable of distinguishing between these three binding times could help making an online specializer self-applicable.

Then, any subject program would be amenable to a wide and well-researched variety of binding-time improvements [3, 25, 14] which would not necessarily change the result of the specialization, but possibly improve specialization efficiency. Since we are really interested in specializing the specializer itself, and generating compilers from interpreters, both of which are typically specialized once and applied many times, we can afford to spend some effort on these improvements. Indeed, this effort has already been done for quite a few applications [16, 14], and turns out to be straightforward for the specializer itself.

4 Reconstructing Binding-Times for the Online Case

The first step towards such a specializer is obviously a suitable binding-time analysis. We have adapted Henglein's constraint-based analysis [12] which is part of a quite a few other offline partial evaluators. Thus, our analysis is type-based, monovariant, and its implementation runs in quasi-linear time using a union-find based algorithm.

We merely show the underlying type system here from which the constraint generation, and the constraint normalization can be derived. Our type language for binding-time types τ is as follows[1]:

$$\tau ::= B \mid \Lambda \mid \Upsilon \mid (\tau_1 \ldots \tau_n) \to \tau_0$$

The binding time B is for static base values, Λ for dynamic values, and Υ for values of unknown binding-time. $(\tau_1 \ldots \tau_n) \to \tau_0$ is for partially static functions

$E \in \mathsf{Expr}, D \in \mathsf{Definition}, \Pi \in \mathsf{Program}$

$$
\begin{array}{rl}
E & ::= V \mid K \mid \texttt{(if } E\ E\ E\texttt{)} \mid \texttt{(}O\ E^*\texttt{)} \mid \texttt{(}P\ E^*\texttt{)} \mid \\
 & \quad \texttt{(let ((}V\ E\texttt{))}\ E\texttt{)} \mid \texttt{(lambda (}V_1 \ldots V_n\texttt{)}\ E\texttt{)} \mid \texttt{(}E\ E\texttt{)} \\
D & ::= \texttt{(define (}P\ V^*\texttt{)}\ E\texttt{)} \\
\Pi & ::= D^+
\end{array}
$$

Fig. 2. Syntax

with argument binding times $\tau_1 \ldots \tau_n$ and result τ_0.

Figure 2 shows the syntax of the language treated by our analysis. It is a side-effect-free higher-order version of Scheme [13]. V denotes variables, K constants, O primitive operators.

$$
\Gamma \vdash K : B \qquad \frac{\Gamma \vdash E_i : B}{\Gamma \vdash \texttt{(}O\ E_1\ \ldots\ E_n\texttt{)} : B} \qquad \frac{\Gamma \vdash E_i : \Lambda}{\Gamma \vdash \texttt{(}O\ E_1\ \ldots\ E_n\texttt{)} : \Lambda}
$$

$$
\frac{\Gamma\{V_i : \tau_i\} \vdash E' : \tau}{\Gamma \vdash \texttt{(lambda (}V_1\ \ldots\ V_n\texttt{)}\ E'\texttt{)} : (\tau_1 \ldots \tau_n) \to \tau} \qquad \frac{\Gamma\{V_i : \Upsilon\} \vdash E' : \Upsilon}{\Gamma \vdash \texttt{(lambda (}V_1\ \ldots\ V_n\texttt{)}\ E'\texttt{)} : \Upsilon}
$$

$$
\frac{\Gamma \vdash E' : (\tau_1 \ldots \tau_n) \to \tau \qquad \Gamma \vdash E_i : \tau_i}{\Gamma \vdash \texttt{(}E'\ E_1\ \ldots\ E_n\texttt{)} : \tau} \qquad \frac{\Gamma \vdash E' : \Upsilon \qquad \Gamma \vdash E_i : \Upsilon}{\Gamma \vdash \texttt{(}E'\ E_1\ \ldots\ E_n\texttt{)} : \Upsilon}
$$

$$
\frac{\Gamma \vdash E_1 : B \qquad \Gamma \vdash E_2 : \tau \qquad \Gamma \vdash E_3 : \tau}{\Gamma \vdash \texttt{(if } E_1\ E_2\ E_3\texttt{)} : \tau} \qquad \frac{\Gamma \vdash E_1 : \Lambda \qquad \Gamma \vdash E_2 : \Lambda \qquad \Gamma \vdash E_3 : \Lambda}{\Gamma \vdash \texttt{(if } E_1\ E_2\ E_3\texttt{)} : \Lambda}
$$

$$
\frac{\Gamma \vdash E_1 : \Upsilon \qquad \Gamma \vdash E_2 : \Upsilon \qquad \Gamma \vdash E_3 : \Upsilon}{\Gamma \vdash \texttt{(if } E_1\ E_2\ E_3\texttt{)} : \Upsilon}
$$

$$
\frac{\Gamma \vdash E : B}{\Gamma \vdash \texttt{(lift } E\texttt{)} : \Upsilon} \qquad \frac{\Gamma \vdash E : \Upsilon}{\Gamma \vdash \texttt{(smash } E\texttt{)} : \Lambda} \qquad \frac{\Gamma \vdash E : \Lambda}{\Gamma \vdash \texttt{(lower } E\texttt{)} : \Upsilon}
$$

$$
\frac{\Gamma \vdash P : (\tau_1 \ldots \tau_n) \to \tau \qquad \Gamma \vdash E_i : \tau_i}{\Gamma \vdash \texttt{(}P\ E_1\ \ldots\ E_n\texttt{)} : \tau}
$$

$$
\frac{\Gamma\{V_i : \tau_i\} \vdash E' : \tau}{\Gamma \vdash \texttt{(define (}F\ V_1\ \ldots\ V_n\texttt{)}\ E'\texttt{)} : (\tau_1 \ldots \tau_n) \to \tau}
$$

$$
\frac{\Gamma\{F_i : \overline{\tau}_i \to \tau_i\} \vdash \Gamma \vdash \texttt{(define (}F_k\ V_1\ \ldots\ V_n\texttt{)}\ E'\texttt{)} : \overline{\tau}_k \to \tau_k}{\Gamma \vdash \mathrm{F}_k : \overline{\tau}_k \to \tau_k}
$$

Fig. 3. Type inference system

Figure 3 shows the type inference system underlying our binding-time analysis. Its judgements have the form $\Gamma :\vdash N : \tau$, where N is either an expression, a procedure name, or a procedure definition. Γ is a set of typing assumptions of the form $N : \tau$. The first rules type expressions, the last two procedure definitions and procedure calls, respectively. $\overline{\tau}$ represents a tuple of binding-time types.

The most obvious difference to offline analyses is the presence of three different lifting operators. Thus, the lifting relation is only a quasi-ordering. Note also that, as opposed to analyses for offline partial evaluators, `lambda` abstractions can either be partially static or receive binding-time Υ (rather than Λ).

The type inference system leads directly to a set of constraints needed to express conditions for a solutions. These constraints in turn are amenable to an efficient algorithm which uses union-find structures, and runs in quasi-linear time [12]. Since

[1] Our analysis also treats partially static data. The extension is straightforward analogous to the handling of functions. Details can be found, for example, in [2].

the constraints as well as the normalization rules are very similar to those presented there, we omit them here for brevity's sake.

Our analysis computes a binding-time annotation for any given program that is maximal with respect to the annotions with binding-time B. Also, it never introduces `smash` operations that could destroy static data, thus preserving all online properties in the specialization phase. Specifically, it allows for `if` expressions with arms of different binding times, which are then both coerced to Υ.

5 Specialization Using the Binding-Time Information

```
(defdata sval
  (static static-val)
  (dynamic dynamic-code)
  (dk dk-val dk-code))

(defdata val
  (scalar scalar-value)
  (closure closure-native closure-label closure-free-svals)
  (ps ps-val-ctor ps-val-args))

(defdata dk-val
  (dk-top)
  (dk-scalar dk-scalar-value)
  (dk-closure dk-closure-native dk-closure-residualize
              dk-closure-label dk-closure-free-svals)
  (dk-ps dk-ps-residualize dk-ps-val-ctor dk-ps-val-args))
```

Fig. 4. Symbolic values

Given a binding-time-annotated subject program, the actual specialization is straightforward: For binding times B and Λ, we employ offline, for Υ we use online techniques. Correspondingly, our symbolic value domain reflects the binding-time distinctions. Figure 4 shows the actual declarations[2] used in our specializer: `Static` stands for partially static data according to the binding-time analysis; its argument is a value of the `val` data type. For `dynamic` values, a code fragment is needed. Only for the `dk` (for "don't know") type, the full information normally present in symbolic values in online specializers is required.

A `static` value may either be a fully static first-order value `scalar`, a closure represented by a procedure `closure-native` that performs an application, a unique label and the symbolic values of its free variables, or a partially static data structure declared by a `defdata` form. The corresponding fields in the `dk-val` type contain additional components `dk-closure-residualize` and `dk-ps-residualize` which are procedures that perform residualization, when applied to symbolic values for their free variables. More on residualization can be found in the next section. A `dk-val` may also be `(dk-top)` which denotes an unknown value; in that case, the `dk-code` field must contain code to compute the value

Figure 5 shows the fragment from the specializer responsible for handling `if` expressions. The main specialization procedure is `specialize-expr`. `AnnIsCond?` tests if the expression to be specialized is a conditional, corresponding selectors extract its components. `AnnExprFetchLevel` extracts the binding time from the

[2] using a special form `defdata` to declare user-defined data types

```
((annIsCond? expr)
 (let* ((test-expr (annFetchCondTest expr))
        (test-sval (specialize-expr test-expr f env cache prg))
        (test-bt (annExprFetchLevel test-expr))
        (then-branch (annFetchCondThen expr))
        (else-branch (annFetchCondElse expr)))
   (cond
    ((equal? 'S test-bt)
     (if (scalar-value (static-val test-sval))
         (specialize-expr then-branch f env cache prg)
         (specialize-expr else-branch f env cache prg)))
    ((equal? '? test-bt)
     (let ((test-val (dk-val test-sval)))
       (cond ((dk-scalar? test-val)
              (if (dk-scalar-value test-val)
                  (specialize-expr then-branch f env cache prg)
                  (specialize-expr else-branch f env cache prg)))
             ((or (dk-closure? test-val) (dk-ps? test-val))
              (specialize-expr then-branch f env cache prg))
             (else ; top
              (let*
                ((then-sval (specialize-expr then-branch f env cache prg))
                 (else-sval (specialize-expr else-branch f env cache prg))
                 (branch-bt (annExprFetchLevel then-branch))
                 (then-code (if (equal? 'D branch-bt)
                                (dynamic-code then-sval)
                                (dk-sval->code then-sval)))
                 (else-code (if (equal? 'D branch-bt)
                                (dynamic-code else-sval)
                                (dk-sval->code else-sval))))
                (dk (dk-top)
                    (annMakeCond (dk-code test-sval)
                                 then-code else-code))))))))
    (else ;; dynamic
     (let ((then-sval (specialize-expr then-branch f env cache prg))
           (else-sval (specialize-expr else-branch f env cache prg)))
       (dynamic (annMakeCond (dynamic-code test-sval)
                             (dynamic-code then-sval)
                             (dynamic-code else-sval))))))))
```

Fig. 5. Specializer fragment for handling `if`

annotated source expression. It can either be `'S` for "static," `'?` for "unknown," and "`'D`" for "dynamic."

It is clearly visible how the branches for `static` and `dynamic` branches directly correspond to code commonly found in offline partial evaluators [14], whereas the case for `dk` code similar to that found in online partial evaluators with a similar domain for symbolic values [19].

Memoization issues also deserve some explanation. Unlike the specializers from the *Fuse* project [30, 19], our specializer generates code directly rather than using an intermediate graph representation. *Fuse* employs the graph representation to defer the questions of whether `let`s should be unfolded until after specialization proper. We employ a simple offline occurrence count analysis analogous to that in offline partial evaluators to decide whether a `let` should be unfolded. This does not always yield optimal code, but considerably simplifies code generation.

To prevent non-termination, a specializer needs to introduce *specialization points*

or *memoization points* [14, 19] which lead to procedures in the residual code. A number of offline and online analyses detect potential loops in residual code unfolding and insert memoization points to break infinite recursion [27, 30, 22, 28, 24]. Our specializer is easily amenable to many of these techniques. For our present implementation, we have chosen the simple strategy to insert memoization points at dynamic `if`s, and at user-introduced program points; this has proven sufficient for most purposes.

On entering a memoization point, the specializer needs to build an entry in a memoization cache indexed by the name of the procedure being specialized, and a projection of the static components of the free variables of the memoization point. Again, for free variables that are `static` or `dynamic`, the code corresponds to that in an offline partial evaluator. For `dk` symbolic values, the code is analogous to the online case. Furthermore, specialization must continue with a new environment composed of *instantiated* versions of the symbolic values of the free variables. Instantiation [19] replaces all dynamic components in symbolic values by fresh variables, which are then formal parameters of the residual procedure being constructed. The code expressions corresponding to the dynamic components being replaced become the actual parameters in the residual call. If partially static data structures are involved, this step performs *arity raising* [18].

After specializing a memoization point, the specializer generates code for it and updates the entry in the memoization cache with the residual procedure constructed from it.

6 Self-Application and Compiler Generation

To achieve successful self-application, it is necesary to ensure that the binding-time analysis will actually annotate the specializer in the desired way. Namely, the following values need to be static to obtain good results:

1. the annotated source program,
2. the expression being specialized, and
3. the names in the current environment.

The first item is trivial to propagate such that the binding-time analysis will recognize it as static. For the second and third item, the online capabilities of the specializer create a slight problem: Unlike in the offline case, closures and constructions of partially static data with non-static binding times are not immediately residualized—rather, the binding time is coerced to Υ then, and the specializer creates `dk` symbolic values that preserve the full information. The specializer effectively postpones the reduce/residualize decision until later.

For residualization of closures, for example, the `dk-closure-label` and `dk-closure-free-svals` fields in the symbolic value would be perfectly sufficient: The label maps to the body of the abstraction in the source program, and specialization could continue on the body with a new environment constructed from the `dk-closure-free-svals-field`. On self-application, however, the matter is not as trivial: The value of the `dk-closure-label` field is dynamic—and this would lead to a dynamic abstraction body, and thereby make the expression argument to the specialization call dynamic with the usual disastrous effects on the result of the binding-time analysis. A similar effect would happen with the environment argument. For `dk-ps` values, the effects are similar.

One way to circumvent the problem would be to perform a common binding-time improvement from the offline world, "The Trick," and loop over all abstractions that could flow into the program point that the specializer is generating code for—or, respectively, all constructors for partially static data. Then specialization could

```
((annIsLambda? expr)
 (let* ((args (annFetchLambdaVars expr))
        (body (annFetchLambdaBody expr))
        (label (annFetchLambdaLabel expr))
        (freevars (annsFetchVars (annFreeVars expr)))
        (free-svals (my-list->list (env-lookup* freevars env))))

   (if (equal? '? (annExprFetchLevel expr))
       (dk
        (dk-closure
         (lambda (arg-svals free-svals)
           (specialize-expr body f
                            (extend-env*
                             (zip-env-by-names freevars free-svals)
                             args (list->my-list-by args arg-svals))
                            cache prg))
         (lambda (free-svals)
           (annMakeLambda
            label args
            (dk-sval->code
             (specialize-expr body f
                              (extend-env*
                               (zip-env-by-names freevars free-svals)
                               args (make-dk-vars args))
                              cache prg))))
         label
         free-svals)
        #f)
       ...)))
```

Fig. 6. Specialization of `dk` abstractions

continue on the found, static, program point. Unfortunately, this approach would require convoluted code and a rudimentary flow analysis.

Instead, we have chosen to move the residualization into an abstraction created at the time of the original specialization of the relevant program point. The abstraction, given the values of the free variables, constructs the residual code. Figure 6 shows the relevant code for specializing `lambda`.

With the residualization abstractions in place, code generation from `dk` values is simple: It is trivial for `dk-top` values (in which case the code is in the `dk-code` field of the symbolic value), and scalars which merely need to be quoted. In the case of abstractions and partially static data, the `dk-closure-residualize` and `dk-ps-residualize` fields carry abstractions that perform the residualization, when applied to the symbolic values of the free variables of the construction. Figure 7 shows the code.

With this single binding-time improvement in place, the specializer successfully self-applies. Somewhat perversely, our specializer is now also amenable to effective *offline* partial evaluation, which achieves the same desired effect, thereby confirming a suspicion by Ruf [19].

7 Experimental Results

We have successfully achieved practical single and double self-application of our online specializer, leading to stand-alone compilers. Moreover, we have successfully

```
(define (dk-sval->code sval)
  (let ((val (dk-val sval)))
    (cond
     ((dk-top? val) (dk-code sval))
     ((dk-scalar? val) (annMakeConst (dk-scalar-value val)))
     ((dk-closure? val)
      (let* ((residualize (dk-closure-residualize val))
             (free-svals (dk-closure-free-svals val)))
        (residualize free-svals)))
     (else ;; dk-ps
      (let* ((residualize (dk-ps-residualize val))
             (free-svals (dk-ps-val-args val)))
        (residualize free-svals))))))
```

Fig. 7. Residualization of `dk` symbolic values

applied Peter Thiemann's *cps-cogen* [26] to the task of partially evaluating our specializer with an *offline* compiler generator. Namely, we have performed preliminary experiments with specializing an interpreter for a first-order functional language. The results are shown in Fig. 8. All timings were run on an IBM PowerPC/250 running Scheme 48 0.39, byte-code implementation of Scheme. The specializer itself measures 2311 `cons` cells, the generated `cogen` a whopping 65432 cells. (*Cogen* generation time was around 5 minutes.) Nevertheless, to our knowledge, this is the first time full self-application of a realistic online partial evaluator has been achieved.

	compiler generation time	compilation time	compiler size
direct	-	68.59	-
compiler	39.93	34.20	12653
cps-cogen	2.24	19.82	6922

Fig. 8. Specializing an interpreter and running a generated compiler (times in seconds, sizes in `cons` cells)

The results demonstrate the practicality of our approach, but also show that more work remains to be done. Presently, offline methods are more effective at partially evaluating our specializer than our specializer itself. Generally, performance could still be better. We see two main areas for improvement:

- Our binding-time analysis presently does not propagate Λ binding times as well as it could according to the type system. The analysis often coerces dynamic values to unknown ones, causing work for the specializer. This is the main reason for the blow-up in size for the compiler generator, and for the performance discrepancy with the offline approach. A variant of the analysis is being developed with promising initial results.
- Much work in the instantiation and residualization routines is still being performed in external primitives which are not being specialized according to the binding-time information present. This is merely a technical matter which should be easy to resolve.

8 Related Work

The main work on realistic online partial evaluation of functional languages has been done by the *Fuse* project at Stanford [30]. The standard reference on online partial

evaluation is still Ruf's Ph.D. thesis [19] which contains an excellent overview of current techniques in online specialization, and detailed assessments of its merits. Mogensen shows how to perform online partial evaluation of the pure λ calculus using higher-order abstract syntax [17]. Glück and Jørgensen have discovered that is possible to automatically generate online specializers from interpreters [7, 8, 11]. The approach has recently been extended to higher-order languages by Thiemann, Glück, and the author [22, 24]. Online specialization techniques beyond constant propagation are supercompilation [29, 10]), a unification-based transformation technique, and generalized partial computation [5], a method that assumes the power of a theorem prover.

Compiler generation and self-application have been among the main fields of interest for researchers in partial evaluation since Futamura's work [4]. This led to the discovery of offline partial evaluation and the construction of practical compilers and compiler generators [15, 16].

As mentioned above, research in the online world has focused on accurate specialization rather than self-application [30]. However, Weise and Ruf [20, 19] argue that compiler generation is desirable with the online approach because of the potential of gaining optimizing compilers which is much more difficult to achieve with the offline approach. However, self-application is not really necessary to achieve compiler generation: Ruf has succeeded in specializing a small first-order online partial evaluator with a full-fledged version of *Fuse* [20, 19]. Still, space requirements are huge and non-practical even for small examples. Ruf already suspected that offline methods might also be suitable to specialize online specializers [19].

Glück [6] has investigated multi-level self-application of online partial evalutors. However, his specializer perform an on-the-fly binding-time analysis which precludes some of the optimizations normally typical of the online approach, such as `if`s with static condition and branches of differing binding times. Also, representation problems blow up the specializations to an impractical extent, which is why Glück and Jørgensen have abandoned the online approach to multi-level specialization in favor of purely offline methods [9].

Mogensen achieves self-application of his online partial evaluator for the pure λ calculus [17] by prescribing the position of the static and dynamic arguments, reducing the reduce/residualize decision to tuple selection.

The pioneering work on efficient binding-time analysis is by Henglein [12]. His type-based approach as well as his fast algorithm for performing the type inference have been used in many offline partial evaluators, most notably Similix [2]. The standard textbook on partial evaluation [14] contains an overview of binding-time improvements.

9 Conclusion

We have shown how to unify online and offline partial evaluation, retaining the specialization accuracy of the online approach, but making compiler generation and self-application possible and and practical. Our main contribution was the use of a binding-time analysis which does not unnecessarily incur a loss of specialization accuracy. In fact, our approach also allows for compiler generation by applying an *offline* partial evaluator to our specializer. We have applied a weaker offline partial evaluator to a stronger online partial evaluator, effectively taking the opposite approach to Ruf's and Weise's work [20, 19] who used a strong specializer to specialize a weaker one. Thus, we have opened the door to the automatic generation of optimizing compilers and compiler generators, which is not easily possible with the offline approach. We believe that our approach is largely orthogonal to both offline and online approaches to increase termination of specialization and specialization

accuracy, such as induction detection, termination analysis, or context propagation. We hope our results may induce more research in hybrid approaches to specialization, and some revived interest in online partial evaluation.

Acknowledgments I am grateful to Peter Thiemann for solid advice, and for his excellent CPS-based offline cogen which provided the front end for the specializer described here. Morry Katz who independently had come up with the same idea motivated me to convert it into practice.

References

1. ACM. *Proc. 1991 ACM Functional Programming Languages and Computer Architecture*, Cambridge, September 1991.
2. Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, July 1993.
3. Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. The essence of eta-expansion in partial evaluation. *Lisp and Symbolic Computation*, 8(1):1–19, 1995.
4. Y. Futamura. Partial evaluation of computation process—an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
5. Yoshihiko Futamura, Kenroku Nogi, and Aki Takano. Essence of generalized partial computation. *Theoretical Computer Science*, 90(1):61–79, 1991.
6. Robert 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).
7. Robert Glück. On the generation of specializers. *Journal of Functional Programming*, 4(4):499–514, October 1994.
8. Robert Glück and Jesper Jørgensen. Generating optimizing specializers. In *IEEE International Conference on Computer Languages*, pages 183–194. IEEE Computer Society Press, 1994.
9. Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions for program specialization. In *Programming Language Implementation and Logic Programming 1995*. Springer-Verlag, 1995. LNCS.
10. Robert Glück and Andrei V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In *Static Analysis*, volume 724 of *Lecture Notes in Computer Science*, pages 112–123. Springer-Verlag, 1993.
11. Robert Glück and Jesper Jørgensen. Generating transformers for deforestation and supercompilation. In B. Le Charlier, editor, *Static Analysis*, volume 864 of *Lecture Notes in Computer Science*, pages 432–448. Springer-Verlag, 1994.
12. Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In FPCA1991 [1], pages 448–472.
13. IEEE. Standard for the Scheme programming language. Technical Report 1178-1990, Institute of Electrical and Electronic Engineers, Inc., New York, 1991.
14. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
15. Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, pages 124–140, Dijon, France, 1985. Springer-Verlag. LNCS 202.
16. Jesper Jørgensen. Compiler generation by partial evaluation. Master's thesis, DIKU, University of Copenhagen, 1991.
17. Torben Æ. Mogensen. Self-applicable online partial evaluation of pure lambda calculus. In Scherlis [21], pages 39–44.
18. Sergei A. Romanenko. Arity raiser and its use in program specialization. In Neil D. Jones, editor, *European Symposium on Programming 1990*, pages 341–360, Copenhagen, Denmark, 1990. Springer-Verlag. LNCS 432.
19. Erik Ruf. *Topics in Online Partial Evaluation*. PhD thesis, Stanford University, Stanford, CA 94305-4055, March 1993. Technical report CSL-TR-93-563.

20. Erik Ruf and Daniel Weise. On the specialization of online program specializers. *Journal of Functional Programming*, 3(3):251–281, July 1993.
21. William Scherlis, editor. *ACM SIGPLAN Symp. Partial Evaluation and Semantics-Based Program Manipulation '95*, La Jolla, CA, June 1995. ACM Press.
22. Michael Sperber, Robert Glück, and Peter Thiemann. Bootstrapping higher-order program transformers from interpreters. In *1996 ACM Symposium on Applied Computing Programming Languages Track*, Philadelphia, 1996. to appear.
23. Michael Sperber and Peter Thiemann. The essence of LR parsing. In Scherlis [21], pages 146–155.
24. Michael Sperber and Peter Thiemann. Realistic compilation by partial evaluation. submitted, 1995.
25. Michael Sperber and Peter Thiemann. Turning an art into a craft: Automatic binding-time improvement. submitted, 1995.
26. Peter Thiemann. Cogen in 6 lines. personal communication.
27. Peter Thiemann. Type-based termination analysis for partial evaluation. Technical Report WSI-95-XX, Universität Tübingen, sep 1995.
28. Peter Thiemann and Robert Glück. The generation of a higher-order online partial evaluator. In Masato Takeichi, editor, *Fuji Workshop on Functional and Logic Programming*, Fuji Susono, Japan, July 1995. World Scientific Press, Singapore.
29. Valentin F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.
30. Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic online partial evaluation. In FPCA1991 [1], pages 165–191.

This article was processed using the LaTeX macro package with the LLNCS document class.