Preliminary Report on a Self-Applicable Online Partial Evaluator for Flowchart

Robert Glück [⋆]

DIKU, Dept. of Computer Science, University of Copenhagen,
DK-2100 Copenhagen, Denmark

Abstract. This is a preliminary report on a self-applicable online partial evaluator for a flowchart language with recursive calls. Self-application of the partial evaluator yields generating extensions that are as efficient as those reported in the literature for offline partial evaluation. This result is remarkable because partial evaluation folklore has indicated that online partial evaluation techniques unavoidably lead to overgeneralized generating extensions. The purpose of this paper is not to argue which line of partial evaluation is better, but to show how the problem can be solved by recursive polyvariant specialization. The online partial evaluator, its techniques and implementation, are presented in a complete way. Full self-application according to the Futamura projections is demonstrated.

1 Introduction

This paper reports on the design and implementation of a self-applicable online partial evaluator for a flowchart language with recursive calls. The partial evaluator does not require partial evaluation techniques that are stronger than those already known, but another organization of the algorithm. This result is remarkable because partial evaluation folklore has indicated that online techniques unavoidably lead to overgeneralized generating extensions [13, Ch. 7]. Offline partial evaluation was invented specifically to solve the problem of self-application [14]. The purpose of this investigation is not to argue which line of partial evaluation is better, but to show how the problem can be solved. Self-application of the online partial evaluator converts interpreters into compilers and produces a self-generating compiler generator, all of which are as efficient as those known from the literature on offline partial evaluation (*e.g.*, [10,13,14,17]).

The offline partial evaluator `mix` for a flowchart language described by Gomard and Jones [10] is well suited as the basis for the online partial evaluator because their partial evaluator does not follow the binding-time annotations of a subject program, but bases its decisions whether to interpret or residualize flowchart commands on a division of the program variables into static and dynamic ones, which was precomputed by a monovariant binding-time analysis. Another important advantage is that partial evaluation for flowchart languages

[⋆] Part of this work was performed while the author was visiting the National Institute of Informatics (NII), Tokyo.

$p ::= \texttt{((}x^*\texttt{) (}l\texttt{) (}b^+\texttt{))}$	(program)
$b ::= \texttt{(}l\ a^*\ j\texttt{)}$	(basic block)
$a ::= \texttt{(}x\ \texttt{:=}\ e\texttt{)}$	(assignment)
$\quad \mid\ \texttt{(}x\ \texttt{:= call}\ l\texttt{)}$	(call)
$j ::= \texttt{(goto}\ l\texttt{)}$	(unconditional jump)
$\quad \mid\ \texttt{(if}\ e\ l\ l\texttt{)}$	(conditional jump)
$\quad \mid\ \texttt{(return}\ e\texttt{)}$	(return)
$e ::= \texttt{(}o\ u^*\texttt{)}$	(simple expression)
$o ::= \texttt{hd} \mid \texttt{tl} \mid \texttt{cons} \mid \texttt{+} \mid \texttt{-} \mid \texttt{=} \mid \texttt{<} \mid \ldots$	(primitive operator)
$u ::= x \mid \texttt{'}v$	(operator argument)
$x \in \mathit{Name} \quad v \in \mathit{Value} \quad l \in \mathit{Label}$	

Fig. 1. Scheme representation of Flowchart programs with recursive calls.

Call

$$\frac{\sigma \vdash_{block} \Gamma(l) \Rightarrow (\langle \mathsf{halt}, v\rangle, \sigma')}{\sigma \vdash_{assign} x\ \texttt{:= call}\ l \ \Rightarrow\ \sigma[x \mapsto v]}$$

Fig. 2. Inference rule extending Hatcliff's operational semantics of `FCL` [11, Fig. 6]

has been very well documented (*e.g.*, [2–4, 10, 11, 13]), which should make our results easily accessible and comparable.

Turning `mix` into an online partial evaluator required two modifications: (1) the division of the program variables is maintained as an updatable set of variable names at specialization time, and (2) the partial evaluator is rewritten to perform *recursive polyvariant specialization* [7] instead of the usual iterative version with an accumulating parameter (pending list). This required an extension of the flowchart language with a simple recursion mechanism. Full self-application according to the Futamura projections is demonstrated by converting an interpreter for Turing-machines [10] into a compiler, a universal parser for regular languages [2] into a parser generator, and the partial evaluator itself into a compiler generator. Self-application of the online partial evaluator can also generate generating extensions that are more optimizing than those produced by the original `mix`. The generating extension of the Ackermann function can specialize and precompute the function at program generation time, thereby producing fully optimized residual programs.

Throughout this paper, we assume that readers are familiar with the basics of partial evaluation, *e.g.*, as presented by Jones *et al.* [13, Part II]. Use is made as much as possible of existing partial evaluation techniques for flowchart languages [10,11] to focus the attention on essential differences, instead of irrelevant details.

```
((m n) (ack)
 ((ack  (if (= m 0) done next))
  (next (if (= n 0) ack0 ack1))
  (done (return (+ n 1)))
  (ack0 (n := 1)
        (goto ack2))
  (ack1 (n := (- n 1))
        (n := (call ack m n))
        (goto ack2))
  (ack2 (m := (- m 1))
        (n := (call ack m n))
        (return n)) ))
```

$$A(m,n) = \begin{cases} n+1 & \text{if } m=0 \\ A(m-1,1) & \text{if } n=0 \\ A(m-1,A(m,n-1)) & \text{otherwise} \end{cases}$$

Fig. 3. Ackermann program and its function

2 A Simple Imperative Language with Recursive Calls

Flowchart is a simple imperative language [10, 11] with variables, assignments, and jumps (Fig. 1). A program consists of a sequence of labeled basic blocks. As is customary, the set of values and labels is that of the Lisp S-expressions. The operational semantics of the language is identical to the one that has been formalized and published by Hatcliff [11], except that we add a simple command for calling blocks:

x `:= call` l

The command executes the block labeled l in a copy of the current store σ and assigns the return value to the variable x in the original store σ. (In an actual implementation this only requires copying the variables that are *live* at the entry of block l.) The command allows recursive calls, has no side-effects, and requires only one additional inference rule in Hatcliff's operational semantics [11, Fig. 6]. The inference rule in Fig. 2 is parameterized with respect to Γ, a partial function that maps labels to blocks in a program, and updates the store σ at x with the return value v, which is marked by halt. Due to its simplicity, the command is easy to interpret and specialize. For the sake of brevity, we refer to the extended language as Flowchart. The syntax can easily be generalized to calls with argument expressions.

An implementation of the Ackermann function using the call command is shown in Fig. 3. For the sake of readability, we annotate each call with the variables that are live at the called block and write integer constants without quotes. The program is defined recursively, takes the non-negative integers `m` and `n` as input, and starts program execution at block `ack`. The double recursion on both `m` and `n` cannot be expressed in terms of primitive recursion.

For the sake of conciseness we shall write many of the programs in pseudocode using constructs such as while ... do and pattern matching case ... of, which are to be regarded as structured ways of writing Flowchart programs. Fig. 4 shows

```
procedure evalpp :: pp × vs × program → value
 block := lookup(pp,program);
 while block is not empty do
 begin command := first-command(block); block := rest(block);
    case command of
    x := rhs:      case rhs of
                   call pp: vs := vs[x ↦ call evalpp(pp,vs,program)];
                   e:       vs := vs[x ↦ evalexp(e,vs)];
    if e pp' pp'': if evalexp(e,vs) = true
                   then block := lookup(pp', program)
                   else  block := lookup(pp'',program);
    goto pp:       block := lookup(pp,program);
    return e:      value := evalexp(e,vs);
 end;
 return value
```

Fig. 4. A self-interpreter for Flowchart written in pseudocode

the *self-interpreter* evalpp for Flowchart written in pseudocode. It inputs a label pp, a store vs, and a program program, and returns the value of the program. Fragments of the interpreted program are written in *italics* for clarity. The self-interpreter makes liberal use of primitive operators such as lookup for finding a block pp in a program and evalexp for evaluating an expression e in a store vs. Note the simplicity of interpreting a call by recursively calling the self-interpreter.

3 Online Partial Evaluation Techniques

A partial evaluator for Flowchart takes values for the static parameters of a program and tries to precompute as many commands as possible. Program code is generated for commands that cannot be precomputed (they are residualized). A specialization strategy is said to be *online* if the values computed at the time of specialization can affect the choice of action taken; otherwise a strategy is said to be *offline* [13, Ch. 4.4.7]. The main advantage of an online strategy is that it can exploit static information that becomes available at specialization time, while an offline strategy bases its decisions on the results of a *binding-time analysis* (BTA) performed before specialization.

We now explain the specialization principles behind the online partial evaluator for Flowchart and in which way it differs from the offline partial evaluator.

3.1 Generating Code for Commands

Fig. 5 shows the specialization of an assignment x := e by an *offline* partial evaluator. The binding time of variable x determines whether the assignment is interpreted or residualized. If x is dynamic, e is reduced to e' by constant folding using the static values in the store vs and an assignment x := e' is generated. If x is static, e is evaluated in vs and the value of x in vs is updated. The congruence

Command	*Done at specialization time*	*Generated code*
$x := e$ (if x is dynamic)	$e' :=$ reduce(e,vs,division)	$x := e'$
$x :=$ *call pp* (if x is dynamic)		$x :=$ *call (pp*,vs*)*
$x := e$ (if x is static)	vs := vs[$x \mapsto$ evalexp(e,vs)]	
$x :=$ *call pp* (if x is static)	vs := vs[$x \mapsto$ **call** evalpp(*pp*,vs,program)]	

Fig. 5. Offline code generation for assignments

Command	*Done at specialization time*	*Generated code*
$x := e$ (if e is dynamic)	$e' :=$ reduce(e,vs,division); division := division $\setminus \{x\}$	$x := e'$
$x :=$ *call pp* (if a var live at *pp* is dynamic)	division := division $\setminus \{x\}$	$x :=$ *call (pp*,vs*)*
$x := e$ (if e is static)	vs := vs[$x \mapsto$ evalexp(e,vs)]; division := division $\cup \{x\}$	
$x :=$ *call pp* (if all vars live at *pp* are static)	vs := vs[$x \mapsto$ **call** evalpp(*pp*,vs,program)]; division := division $\cup \{x\}$	

Fig. 6. Online code generation for assignments

Command	*Done at specialization time*	*Generated code*
if e pp' pp" (if e is dynamic)	$e' :=$ reduce(e,vs,division)	*if e' (pp',vs) (pp",vs)*
if e pp' pp" (if e is static and value = true)	value := evalexp(e,vs)	*goto (pp'*,vs*)*
if e pp' pp" (if e is static and value = false)	value := evalexp(e,vs)	*goto (pp"*,vs*)*
goto pp		*goto (pp*,vs*)*
return e	$e' :=$ reduce(e,vs,division);	*return e'*

Fig. 7. Online and offline code generation for control flow (no transition compression)

of the binding times calculated by the BTA guarantees that e can be evaluated in vs whenever x is static [12]. In the case of `mix`, which uses a monovariant BTA, the same division is valid at all program points. The division can be represented by a set division that contains the names of all static variables. Variables that are not in this set are dynamic.

Likewise, x := call pp is interpreted or residualized depending on the binding time of x. If x is dynamic, the call is residualized as x := call (pp,vs) and later a residual block (pp,vs), that is block pp specialized with respect to vs, is generated. Otherwise, the call is static and interpreted by the self-interpreter evalpp (Fig. 4).

In an *online* partial evaluator the division is not known until specialization time. Thus, instead of using the binding time of x in an assignment x := e, the binding time of e determines whether the assignment is interpreted or residualized. The binding time of x is changed accordingly by adding it to or removing it from the division. Fig. 5 shows the partial evaluation of an assignment by an online partial evaluator. The binding times of variables are determined and propagated at specialization time and may affect the course of partial evaluation.

Similarly for x := call pp, except that the binding times of all variables that are *live* at pp determine whether the assignment is interpreted or residualized. If all live variables are static, the call is interpreted in vs and the return value is assigned to x. Otherwise, an assignment x := call (pp,vs) is generated. The division is updated according to the binding time of x.

In an online partial evaluator the binding time of a variable x on the left-hand side of an assignment depends on the binding time of the expression or the call on the right-hand side. Depending on the outcome, the division is updated at specialization time. The division is not predetermined and can change during partial evaluation.

Control flow commands Code generation without transition compression is shown in Fig. 7. The partial evaluation of control flow commands is the same in online and offline partial evaluation. However, as a consequence of the online treatment of assignments, the decision taken by an online partial evaluator, namely whether to residualize a conditional or to select one of the branches (pp', pp''), depends on the actual static values obtained from the assignments preceding the conditional. Even though the only visible difference between code generation in an online and offline partial evaluator is confined to the handling of assignments (Fig. 5 *vs.* Fig. 6), the process of partial evaluation proceeds quite differently.

Transition compression Residual programs produced by the code generation described above often contain chains of trivial transitions (blocks consisting only of goto commands), which makes them less readable. Transition compression can be performed during partial evaluation by continuing code generation directly with the commands in a block pp instead of generating goto pp (Fig. 7). Transition compression can also be performed after partial evaluation in a post-processing phase. The choice of the transition compression strategy does not affect the specialization strength or the self-applicability of the partial evaluators, only the size of the residual programs which they generate.

3.2 Two Simple Polyvariant Specialization Algorithms

Program point specialization incorporates the values of the static variables into a program point. In *polyvariant block specialization* [2] each block in a subject

```
procedure polyloop :: pp × vs → code
  pend := {(pp,vs)}; done := {}; code := {};
  while pend is not empty do
  begin
    pick (pp,vs) ∈ pend;
    code := code ∪ generate residual block for pp using the values in vs;
    done := done ∪ {(pp,vs)}; pend := (pend ∪ successors(pp,vs)) \ done
  end;
  return code
```

Fig. 8. A simple iterative specialization algorithm polyloop [10]

```
procedure polyrec :: pp × vs × code → code
  if no residual block labeled (pp,vs) exists in code then
  begin
    code := code ∪ generate residual block for pp using the values in vs;
    let {(pp_1,vs_1), ... , (pp_n,vs_n)} = successors(pp,vs);
    code := call polyrec(pp_1,vs_1,code);
    ...
    code := call polyrec(pp_n,vs_n,code)
  end;
  return code
```

Fig. 9. A simple recursive specialization algorithm polyrec [7]

program may be specialized with respect to several different static stores. A residual block labeled (pp,vs) is the concatenation of the code generated for the commands in block pp using the values in vs. Polyvariant block specialization can be implemented in two different ways. Traditionally, an iterative method is used that maintains a set of pending and done specialization tasks. We shall see that a recursive method enables self-application of the online partial evaluator.

1. The *iterative method* in Fig. 8 maintains two sets, pend and done, to keep track of the pairs (pp,vs) that are pending specialization and those for which a residual block was already generated. Block specialization is repeated by the while-loop until pend is empty. After generating a residual block, the set of successor blocks, successors(pp,ss) = $\{(\mathsf{pp}_1,\mathsf{vs}_1)$, ..., $(\mathsf{pp}_n,\mathsf{vs}_n)\}$, that is all blocks that occur in conditionals and calls of the residual block (pp,vs), are added to pend as new specialization tasks, unless they are already in done. The residual blocks are collected in a set code and returned as the final result. Iterative block specialization is invoked by **call** polyloop $(\mathsf{pp}_0,\mathsf{vs}_0)$, where pp_0 is the initial label of the subject program and vs_0 is the initial static store.
2. The *recursive method* in Fig. 9 performs block specialization in a depth-first manner without maintaining a set pend. The successor blocks are specialized *immediately* after the specialization of a residual block is completed. The set code of generated residual blocks doubles as set done. A block specialization

(pp,vs) is only performed if the residual block for (pp,vs) does not yet exist in code. Set code is single-threaded through the recursive calls to avoid repeating the generation of the same residual block. Consider as an example the specialization of the two successors blocks (pp_1,vs) and (pp_2,vs) that occur in a residual conditional if e (pp_1,vs) (pp_2,vs):

```
code := call polyrec(pp1,vs,code);
code := call polyrec(pp2,vs,code)
```

Recursive block specialization is invoked by **call** polyrec (pp_0,vs_0,{}), where pp_0 is the initial label, vs_0 is the initial static store, and {} is the initially empty set of residual blocks.

While the iterative version makes use of a data structure (pend) to keep track of the specialization tasks, the recursive version relies on the call stack of the implementation language. The iterative version corresponds to a tail-recursive function where pend is an accumulating parameter. Functions with accumulating parameters are notorious for being difficult to specialize.

3.3 Specializing the Simple Specialization Algorithms

The challenge of self-application is the specialization of the partial evaluation algorithm by itself with respect to a known subject program and a known division, but unknown values for the static store.

Consider first the iterative version. The set pend contains pairs of the form (pp,vs) where pp is part of the subject program to which the partial evaluator is specialized and vs is unknown. As a result, pend becomes dynamic. In offline partial evaluation this problem was solved by precomputing the static set and using a binding-time improvement, called "The Trick" [13], because the static set contains finitely many components (pp,names-in-vs). The lookup of a block pp in the subject program is implemented such that the dynamic pp is compared to all possible values it can assume and specializes the block to all possible outcomes. This trick is necessary to avoid generating trivial generating extensions in which no specialization is performed. The set is precomputed by a BTA.

Now, consider the recursive version. The problem of losing the control information does not arise because there is no set pend in which the program points pp get dynamized. This depth-first specialization method enables the information to be propagated statically and accurately, even in the case of self-application. This is the method that we choose for the implementation of the self-applicable online partial evaluator. It has the additional advantage that it can be used regardless of whether code generation for commands is online or offline. It can thus be used in self-applicable online and offline partial evaluators.

4 An Algorithm for Online Partial Evaluation

We now present the complete algorithm for online partial evaluation based on the specialization techniques described above. It performs code generation with

```
procedure onmix :: program × division × vs → code
 pp   := initial-label(program);
 code := make-header(varlist(program,division),pp,vs);
 return call pepoly(pp,vs,division,program,code)

procedure pepoly :: pp × vs × division × program × code → code
 if ¬done(pp,vs,code)
 then begin
   code  := new-block(code,pp,vs);
   block := lookup(pp,program);
   while block is not empty do
   begin command := first-command(block); block := rest(block);
     case command of
     x := rhs:     if static(vars(rhs,program),division)
                   then begin
                          case rhs of
                            call pp: vs := vs[x ↦ call evalpp(pp,vs,program)];
                            e:       vs := vs[x ↦ evalexp(e,vs)];
                          division := division ∪ {x} end
                   else begin
                         case rhs of
                           call pp: code := call pepoly(pp,vs,division,program,code);
                                    code := add-to(code,make-asg(x,make-call(pp,vs)));
                           e:       code := add-to(code,make-asg(x,reduce(e,vs,division)));
                         division := division \ {x} end;
     if e pp' pp'': if static(e,division)
                    then if evalexp(e,vs) = true
                           then block := lookup(pp', program)
                           else  block := lookup(pp'',program)
                    else begin
                      code := call pepoly(pp', vs,division,program,code);
                      code := call pepoly(pp'',vs,division,program,code);
                      code := add-to(code,make-if(reduce(e,vs,division),(pp',vs),(pp'',vs)))
                   end;
     goto pp:      block := lookup(pp,program);
     return e:     code  := add-to(code,make-return(reduce(e,vs,division)));
   end (* while *)
 end; (* then *)
 return code
```

Fig. 10. The self-applicable algorithm for online partial evaluation of Flowchart

on-the-fly transition compression and recursive polyvariant block specialization. The algorithm is given in pseudocode in Fig. 10.

The input to the main procedure onmix is a subject program program, a division division of the parameters of the subject program, and a store vs containing the values of the static parameters. The residual program code is returned as out-

```
((n) (ack-0)
 ((ack-0  (if (= n 0) ack0-0 ack1-0)) (ack-1  (if (= n 0) ack0-1 ack1-1))
  (ack0-0 (return 3))  ; A(1,1) = 3   (ack0-1 (return 2))  ; A(0,1) = 2
  (ack1-0 (n := (- n 1))              (ack1-1 (n := (- n 1))
          (n := (call ack-0 n))               (n := (call ack-1 n))
          (n := (call ack-1 n))               (n := (call ack-2 n))
          (return n))                         (return n))
                                      (ack-2  (return (+ n 1))) ))
```

Fig. 11. Ackermann program specialized with respect to m=2

put. Some initializations are performed before invoking pepoly, which performs the actual specialization of the blocks in the subject program.

The procedure pepoly implements recursive polyvariant block specialization. Block pp is specialized with respect to vs and division if no residual block exists in code, which is tested by the primitive operator done. If the residual block already exists, code is returned unchanged. Otherwise, the block is fetched from the program by lookup and the header of the new block is added to code. The list of commands is then specialized starting with the first command until it is empty. The while-loop contains a case dispatch which generates code for the commands as described above. The code is generated with transition compression as can be seen in the case of specializing goto pp. For example, see the case of goto pp: instead of generating a residual jump goto (pp,vs), specialization continues at pp. The primitive operation vars(rhs,program) in the assignment case returns e, if rhs = e, or an expression with the variables that are live at pp, if rhs = call pp.

The program in Fig. 11 is an example of a residual program produced by the online partial evaluation algorithm. It was obtained by specializing the Ackermann program with respect to m = 2. No post-optimization was performed on the residual program. Note that m is static throughout the entire Ackermann program (Fig. 3), while variable n, which is initially dynamic, becomes static in block ack0 due to assignment n := 1 and, consequently, the call to ack in ack2 is static and can be precomputed at specialization time. The result are the residual blocks ack0-0 and ack0-1 which return a constant. The same call to ack is partially static if it is reached from ack1 where n is dynamic. The offline partial evaluator mix [10] cannot perform this specialization (this would require a polyvariant expansion of the original Ackermann program based on a polyvariant binding-time analysis before applying mix). A post-optimization of the residual program could replace (call ack-2 n) in ack1-1 by expression (+ n 1).

The online partial evaluator described in this paper has the specialization strength of an offline partial evaluator with a polyvariant binding-time analysis and monovariant return values. This functional equivalence is not surprising because an offline partial evaluator with a maximally polyvariant binding-time analysis can be as accurate as an online partial evaluator [3].

4.1 Live Static Variables

The specialization of blocks with respect to dead static variables can cause considerable code duplication in the residual programs [10]. This is even more critical in an online partial evaluator because the division and the static store can grow and shrink during partial evaluation. It is therefore essential to remove all dead static variables from division and vs each time a new block is looked up in the subject program. At the beginning of onmix before pepoly is called, the set of live variables is determined for each block entry in a subject program and a parameter containing this information is added to pepoly. For readability this parameter and the operations restricting division and vs to the live variables after each lookup were omitted in Fig. 10. However, these cleaning-up operations are crucial for reducing the size of the generated residual program.

4.2 Self-Application

The classification of the three parameters of the main procedure onmix: program and division are static and vs is dynamic. Operations that depend only on program and division can be static, while all other operations that may depend on vs are dynamic. In particular, the parameters pp, division and program remain static and only vs and code are dynamic. The recursive method of polyvariant block specialization keeps this essential information static (pp, division, program), providing the key to successful self-application. Assignments that depend only on static variables are fully evaluated when the partial evaluator us specialized with respect to a subject program and do not occur in the generating extensions produced by self-application. As an example, the important tests static are always static when the online partial evaluator is specialized. They will thus never occur in the generating extensions. Also, a change of the transition compression strategy does not affect the binding-time separation.

5 Specializing the Online Partial Evaluation Algorithm

A classic example is the specialization of a partial evaluator with respect to an interpreter for Turing machines, which yields a compiler from Turing-machine programs to the residual language of the partial evaluator, here Flowchart. We used the same Turing-machine program and the same Turing-machine interpreter[1] written in Flowchart as in publication [10, Fig. 3 and Fig. 4].

The *first Futamura projection* translates the Turing-machine program p into a Flowchart program tar by specializing Turing-machine interpreter int with respect to p by the online partial evaluator onmix:

$$\mathtt{tar} = [\![\mathtt{onmix}]\!]\,\mathtt{(int,p)}. \qquad (1)$$

[1] A generalization operator was inserted to change the classification of the variable representing the left-hand side of the tape from static to dynamic at `Left := (GEN '())`.

The program tar is identical to the one produced by fcl-mix [10, Fig. 5] modulo minor syntactic differences between the flowchart languages.

The *second Futamura projection* yields a compiler comp by self-application of onmix:

$$\mathtt{comp} = [\![\mathtt{onmix}]\!]\,\mathtt{(onmix, int)}. \tag{2}$$

The compiler comp translates Turing-machine programs into Flowchart. The compiler is as efficient as the one presented in [10, App. II]. Compilation is done recursively instead of iteratively due to the recursive polyvariant specialization used in the specialized onmix. This is also an example how the structure of the generated compilers can be influenced by specializing different partial evaluators.

The *third Futamura projection* yields a compiler generator cogen by double self-application:

$$\mathtt{cogen} = [\![\mathtt{onmix}]\!]\,\mathtt{(onmix, onmix)}. \tag{3}$$

The compiler generator is as efficient as the one reported for mix [10], except that cogen performs recursive polyvariant specialization and produces generating extensions that also perform recursive polyvariant specialization. A compiler generator produced by the third Futamura projection must be *self-generating*, which is a necessary condition for its correctness [6], and so is cogen, which produces a textually identical copy of itself when applied to onmix:

$$\mathtt{cogen} = [\![\mathtt{cogen}]\!]\,\mathtt{onmix}. \tag{4}$$

Applying cogen to the Ackermann program ack yields a generating extension ackgen, which produces residual programs of ack given a value for m, such as the one shown in Fig. 11.

$$\mathtt{ackgen} = [\![\mathtt{cogen}]\!]\,\mathtt{ack}. \tag{5}$$

Application of cogen to Bulyonkov's universal parser for regular languages [2] yields a parser generator parsegen that is comparable to the one reported in [7, Fig. 5], if we disregard the fact that the one in this paper is produced by self-application of onmix (or by cogen) and implemented in Flowchart, while the one produced by quasi-self-application [7] is implemented in Scheme and arity raised by the postprocessor of Unmix [17].

5.1 Overview of Performance

Tables 1 and 2 show some of the preliminary running times for a version of the online specialization algorithm onmix. Program p and interpreter int are the Turing-program and the Turing interpreter [10]. The data for ack is $\mathtt{m} = 2$ and $\mathtt{n} = 3$. The running times are measured AMD Athlon cpu milliseconds using Dr. Scheme version 4.1.3 under Windows XP Home Edition 2002 and include garbage collection, if any. The running times are comparable to those reported in the literature, albeit *all* Turing-related ratios are slightly smaller compared to the results [10, Tab. 2], which were reported for a different hardware/software.

Run		Time	Ratio
`out`	$= [\![$`int`$]\!]$ `(p, d)`	78	
	$= [\![$`tar`$]\!]$ `d`	16	4.9
`tar`	$= [\![$`onmix`$]\!]$ `(int, p)`	172	
	$= [\![$`comp`$]\!]$ `p`	47	3.7
`comp`	$= [\![$`onmix`$]\!]$ `(onmix, int)`	938	
	$= [\![$`cogen`$]\!]$ `int`	547	1.7
`cogen`	$= [\![$`onmix`$]\!]$ `(onmix, onmix)`	3016	
	$= [\![$`cogen`$]\!]$ `onmix`	907	3.3

Table 1. Turing interpreter

Run		Time	Ratio
`out`	$= [\![$`ack`$]\!]$ `(m, n)`	20	
	$= [\![$`ackm`$]\!]$ `n`	4	5
`ackm`	$= [\![$`onmix`$]\!]$ `(ack, m)`	168	
	$= [\![$`ackgen`$]\!]$ `m`	24	7
`ackgen`	$= [\![$`onmix`$]\!]$ `(onmix, ack)`	668	
	$= [\![$`cogen`$]\!]$ `ack`	496	1.3

Table 2. Ackermann program

6 Related Work

Conventional wisdom holds that only offline partial evaluators using a binding-time analysis can be specialized into efficient generating extensions (*e.g.*, [1] and [13, Ch. 7]). Offline partial evaluation was invented specifically to solve this problem. Mix was the first efficiently self-applicable partial evaluator [14].

The self-applicable partial evaluator presented in this paper makes use of recursive polyvariant specialization [7] to ensure that the information needed for specialization of blocks is not prematurely lost (dynamized) at program generator generation time. An implementation of recursive polyvariant specialization by the author in 1993 is part of the Unmix distribution, an offline partial evaluator for a first-order subset of Scheme [17].

A *higher-order* pending list was used by a breadth-first inverse interpreter to allow good specialization by the offline partial evaluator Similix [8, p. 15], but requires a partial evaluator powerful enough to specialize higher-order values. The self-application of an online partial evaluator for the λ-calculus without polyvariant specialization was reported, but the compilers were of the "overly general" kind [15]. A compromise strategy to self-application of online partial evaluators is a hybrid 'mixline' approach to partial evaluation that distinguishes between static, dynamic, and unknown binding times [13, Ch. 7.2.3] and [19,20]. A notable exception on the self-application of online specializers is V-Mix [5] and [9]. A weaker online specializer was specialized by a stronger one [18].

7 Conclusions and Future Work

We showed that not only offline partial evaluators, but also online partial evaluators can yield generating extensions by self-application that are as efficient as those reported in the literature for offline partial evaluation. It is noteworthy that this did not require partial evaluation techniques that are stronger than those already known today, only a restructuring of the partial evaluator. Although the design of the algorithm is based on a number of existing techniques, their combination in a new and non-trivial way produced this synergetic effect.

Full self-application according to the Futamura projections was demonstrated by implementing a non-trivial online partial evaluator for a flowchart language extended with a simple recursion mechanism. Self-application produced generating extensions whose structure is as “natural and understandable” as in the case of offline partial evaluation [16]. There was no loss of efficiency and no overgeneralization. Self-application of the online partial evaluator can also lead to generating extensions that are more optimizing than those produced by the offline partial evaluators for the flowchart language, such as the generating extension of the Ackermann function. The algorithm for online partial evaluation, the design, techniques and demonstration, were presented in a complete and transparent way. Several attempts have been made previously, including work by the author. We believe that the online partial evaluator for the flowchart language presented in this paper provides the clearest solution to date. It is believed that the techniques presented here can be carried over to other recursive programming languages. It is hoped that this investigation into self-application can be a basis for novel partial evaluators and stronger generating extensions.

Acknowledgements It is a great pleasure to thank Akihiko Takano for providing excellent working conditions to the author at the National Institute of Informatics (NII), Tokyo. Thanks to Kenji Moriyama for help with the program measurements and to Neil Jones and the anonymous reviewers for insightful comments.

References

1. A. Bondorf, N. D. Jones, T. Æ. Mogensen, P. Sestoft. Binding time analysis and the taming of self-application. Research note, DIKU, Dept. of Computer Science, University of Copenhagen, 1988.
2. M. A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21(5):473–484, 1984.
3. N. H. Christensen, R. Glück. Offline partial evaluation can be as accurate as online partial evaluation. *ACM TOPLAS*, 26(1):191–220, 2004.
4. S. Debois. Imperative-program transformation by instrumented-interpreter specialization. *Higher-Order and Symbolic Computation*, 21(1-2):37–58, 2008.
5. R. Glück. Towards multiple self-application. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 309–320. ACM Press, 1991.

6. R. Glück. Is there a fourth Futamura projection? In *Partial Evaluation and Program Manipulation. Proceedings*, 51–60. ACM Press, 2009.
7. R. Glück. An experiment with the fourth Futamura projection. In A. Pnueli, I. Virbitskaite, A. Voronkov (eds.), *Perspectives of System Informatics. Proceedings*, LNCS 5947, 135–150. Springer-Verlag, 2010.
8. R. Glück, Y. Kawada, T. Hashimoto. Transforming interpreters into inverse interpreters by partial evaluation. In *Partial Evaluation and Semantics-Based Program Manipulation. Proceedings*, 10–19. ACM Press, 2003.
9. R. Glück, V. F. Turchin. Application of metasystem transition to function inversion and transformation. In *Proceedings of the Int. Symposium on Symbolic and Algebraic Computation (ISSAC'90)*, 286–287. ACM Press, 1990.
10. C. K. Gomard, N. D. Jones. Compiler generation by partial evaluation: a case study. *Structured Programming*, 12(3):123–144, 1991.
11. J. Hatcliff. An introduction to online and offline partial evaluation using a simple flowchart language. In J. Hatcliff, T. Æ. Mogensen, P. Thiemann (eds.), *Partial Evaluation. Practice and Theory*, LNCS 1706, 20–82. Springer-Verlag, 1999.
12. N. D. Jones. Automatic program specialization: a re-examination from basic principles. In D. Bjørner, A. P. Ershov, N. D. Jones (eds.), *Partial Evaluation and Mixed Computation*, 225–282. North-Holland, 1988.
13. N. D. Jones, C. K. Gomard, P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
14. N. D. Jones, P. Sestoft, H. Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In J.-P. Jouannaud (ed.), *Rewriting Techniques and Applications*, LNCS 202, 124–140. Springer-Verlag, 1985.
15. T. Æ. Mogensen. Self-applicable online partial evaluation of the pure lambda calculus. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 39–44. ACM Press, 1995.
16. S. A. Romanenko. A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In D. Bjørner, A. P. Ershov, N. D. Jones (eds.), *Partial Evaluation and Mixed Computation*, 445–463. North-Holland, 1988.
17. S. A. Romanenko. The specializer Unmix, 1990. Program and documentation available from `ftp://ftp.diku.dk/pub/diku/dists/jones-book/Romanenko/`.
18. E. Ruf, D. Weise. On the specialization of online program specializers. *Journal of Functional Programming*, 3(3):251–281, 1993.
19. M. Sperber. Self-applicable online partial evaluation. In O. Danvy, R. Glück, P. Thiemann (eds.), *Partial Evaluation. Proceedings*, LNCS 1110, 465–480. Springer-Verlag, 1996.
20. E. Sumii, N. Kobayashi. Online-and-offline partial evaluation: a mixed approach. In *Partial Evaluation and Semantics-Based Program Manipulation. Proceedings*, 12–21. ACM Press, 2000.

A Appendix: Ackermann Generation Extension

Fig. 12 shows the complete generating extension of the Ackermann program in Fig. 3 produced by self-application of the online partial evaluator `onmix`. Given a value for `m`, the generating extension produces a residual program such as the one shown in Fig. 11. No arity raising was performed, so the input to the generating extension is a list `initial-vs` that contains a single value (`m`). The generating

```
((initial-vs) (0-0)
 ((0-0  (vs    := (initstore '(m) initial-vs))
        (vs    := (procrustes vs '(m)))
        (code  := (make-header '(n) 'ack vs))
        (code  := (call 1-0 vs code))
        (return code))

  (1-0  (if (done 'ack vs code) 2-0 3-0))
  (2-0  (return code))
  (3-0  (code  := (new-block code 'ack vs))
        (if (evalexp '(= m 0) vs) 16-0 17-0))
  (16-0 (return (add-to code (list 'return (reduce '(+ n 1) vs '())))))
  (17-0 (vs1   := (procrustes vs '(m)))
        (vs2   := (procrustes vs '(m)))
        (cond  := (list 'if (reduce '(= n 0) vs '(m)) (cons 'ack0 vs1) (cons 'ack1 vs2)))
        (vs    := vs1)
        (code  := (call 1-1 vs code))
        (vs    := vs2)
        (code  := (call 1-2 vs code))
        (return (add-to code cond)))

  (1-1  (if (done 'ack0 vs code) 2-0 3-1))
  (3-1  (code  := (new-block code 'ack0 vs))
        (vs    := (assign 'n (evalexp '1 vs) vs))
        (vs    := (assign 'm (evalexp '(- m 1) vs) vs))
        (value := (call 50-0 vs))
        (vs    := (assign 'n value vs))
        (return (add-to code (list 'return (reduce 'n vs '(m n))))))

  (1-2  (if (done 'ack1 vs code) 2-0 3-2))
  (3-2  (code  := (new-block code 'ack1 vs))
        (code  := (add-to code (list 'n ':= (reduce '(- n 1) vs '(m)))))
        (vs1   := vs)
        (vs    := (procrustes vs '(m)))
        (code  := (call 1-0 vs code))
        (code  := (add-to code (list 'n ':= (make-call 'ack vs '(n)))))
        (vs    := vs1)
        (vs    := (assign 'm (evalexp '(- m 1) vs) vs))
        (vs1   := vs)
        (vs    := (procrustes vs '(m)))
        (code  := (call 1-0 vs code))
        (code  := (add-to code (list 'n ':= (make-call 'ack vs '(n)))))
        (vs    := vs1)
        (return (add-to code (list 'return (reduce 'n vs '(m))))))

  (50-0 (if (evalexp '(= m 0) vs) 57-0 58-0))
  (57-0 (return (evalexp '(+ n 1) vs)))
  (58-0 (if (evalexp '(= n 0) vs) 57-1 58-1))
  (57-1 (vs    := (assign 'n (evalexp '1 vs) vs))
        (vs    := (assign 'm (evalexp '(- m 1) vs) vs))
        (value := (call 50-0 vs))
        (vs    := (assign 'n value vs))
        (return (evalexp 'n vs)))
  (58-1 (vs    := (assign 'n (evalexp '(- n 1) vs) vs))
        (value := (call 50-0 vs))
        (vs    := (assign 'n value vs))
        (vs    := (assign 'm (evalexp '(- m 1) vs) vs))
        (value := (call 50-0 vs))
        (vs    := (assign 'n value vs))
        (return (evalexp 'n vs))) ))
```

Fig. 12. Generating extension of the Ackermann program (`m` *static*, `n` *dynamic*)

extension inherits several primitive operations from the actual implementation of onmix. The store vs is updated by primitive operation assign and the primitive operation procrustes limits a store vs to the bindings of the variables listed as its second argument. Code is generated and added to the residual program by the primitive operations make-header, make-call, add-to, and new-block. The primitive operations reduce and evalexp are identical to the ones in Sect. 3.1.

The blocks from 1-0 to 17-0, from 1-1 to 3-1, and from 1-2 to 3-2 produce residual versions of the original blocks ack, ack0, and ack1, respectively. They are specialized versions of procedure pepoly (Fig. 10) implemented in Flowchart.

The blocks from 50-0 to 58-1 are a complete implementation of the Ackermann function albeit with interpretive overhead inherited from the self-interpreter evalpp (Fig. 4), which is part of onmix. The entry block 50-0 is called in block 3-1 to compute the value of the Ackermann function. It is instructive to compare this implementation to the original program (Fig. 3). Even though this version is slower than the original program, it allows the generating extension to precompute the Ackermann function when generating a residual program.

Transition compression has duplicated some commands (*e.g.*, the commands of the original block ack2 are the last four commands of the blocks 57-1 and 58-1).