On the Specialization of Online Program Specializers*

Erik Ruf and Daniel Weise

Technical Report: CSL-TR-92-534
(also FUSE Memo 92-13)

July, 1992

Computer Systems Laboratory
Departments of Electrical Engineering & Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

Program specializers improve the speed of programs by performing some of the programs' reductions at specialization time rather than at runtime. This specialization process can be time-consuming; one common technique for improving the speed of the specialization of a particular program is to specialize the specializer itself on that program, creating a custom specializer, or *program generator*, for that particular program.

Much research has been devoted to the problem of generating *efficient* program generators, which do not perform reductions at program generation time which could instead have been performed when the program generator was constructed. The conventional wisdom holds that only *offline* program specializers, which use *binding time annotations*, can be specialized into such efficient program generators. This paper argues that this is not the case, and demonstrates that the specialization of a nontrivial *online* program specializer similar to the original "naive MIX" can indeed yield an efficient program generator.

The key to our argument is that, while the use of *binding time information* at program generator generation time is necessary for the construction of an efficient custom specializer, the use of explicit *binding time approximation* techniques is not. This allows us to distinguish the problem at hand (*i.e.,* the use of binding time information during program generator generation) from particular solutions to that problem (*i.e.,* offline specialization). We show that, given a careful choice of specializer data structures, and sufficiently powerful specialization techniques, binding time information can be inferred and utilized without the use of explicit binding time approximation techniques. This allows the construction of efficient, optimizing program generators from online program specializers.

Key Words and Phrases: Partial Evaluation, Program Specialization, Self-Application, Program Generation, Online Specialization, Offline Specialization, Binding Time Information, Binding Time Analysis.

*This research has been supported in part by NSF Contract No. MIP-8902764, and in part by Advanced Research Projects Agency, Department of Defense, Contract No. N0039-91-K-0138. Erik Ruf is funded by an AT&T Foundation Ph.D. Scholarship.

Introduction

A *program specializer* (also called a *partial evaluator*) transforms a program and a specification restricting the possible values of its inputs into a *specialized* program that operates only on those input values satisfying the specification. The specializer uses the information in the specification to perform some of the program's computations at specialization time, producing a specialized program that performs fewer computations at runtime, and thus runs faster than the original program.

Program specialization is useful when some of a program's inputs remain constant over several executions of the program. If we specialize the program on those constant values, the specialized program can be applied repeatedly to the non-constant values, allowing the cost of specialization to be amortized across the repeated executions of the specialized program. For example, instead of repeatedly executing an interpreter on the same program, but different inputs:

```
result-1 := interpreter(program-1,inputs-1)
result-2 := interpreter(program-1,inputs-2)
...
result-3 := interpreter(program-1,inputs-n)
```

we can specialize the interpreter on the program, and use the specialized program several times. Assuming the existence of a two-input procedure, `specializer`, which takes a program and a description of its constant inputs, and produces a program specialized on those inputs, we could instead execute the sequence

```
specialized-interpreter := specializer(interpreter,program-1)
result-1 := specialized-interpreter(inputs-1)
result-2 := specialized-interpreter(inputs-2)
...
result-3 := specialized-interpreter(inputs-n)
```

The number of executions necessary to repay the cost of specialization depends upon the degree to which specialization improves the speed of the program, and upon the cost of running the specializer. The quality of specialization has been improved by various techniques, while the efficiency of specialization has been addressed primarily by means of self-application; that is, specializing the specializer itself.[1]

The idea of improving the efficiency of specialization by specializing the specializer, independently discovered by Futamura [19] and Ershov [18], is based on the same observation we made above. That is, if a program is executed repeatedly on a constant input, we can benefit by specializing the program on that constant input, and executing the specialized program instead. In many cases, the specializer itself is executed repeatedly on a constant input, namely the program to be specialized. For example, an interpreter may be specialized on different programs, as in

```
specialized-interpreter-1 := specializer(interpreter-1,program-1)
specialized-interpreter-2 := specializer(interpreter-1,program-2)
...
specialized-interpreter-n := specializer(interpreter-1,program-3)
```

[1]Several other approaches include handwriting a specializer generator [23, 28] and performing more operations statically prior to specialization time [12]. This paper addresses only the self-application technique.

which can be replaced by the sequence

```
specialized-specializer := specializer(specializer,interpreter-1)²
specialized-interpreter-1 := specialized-specializer(program-1)
specialized-interpreter-2 := specialized-specializer(program-2)
...
specialized-interpreter-3 := specialized-specializer(program-n)
```

which is more efficient. This use of the specializer to improve itself is called *self-application*, and the specialized specializer is often referred to as a *program generator*. The generation of efficient program generators has motivated much of the recent research in program specialization.

All that is required for self-application is that the specializer be written in the same language as the programs it processes, so that it can treat itself as a program to be specialized. This *auto-projection* property, although sufficient to allow self-application to be performed, is insufficient to guarantee efficient results. That is, when the specialized specializer runs, it may perform reductions which instead could have been performed once, when the specializer was specialized.

This desire to construct efficient program generators motivated the invention of *offline* program specialization[31], in which all of the specializer's reduce/residualize decisions are made prior to specialization time, usually via an automatic prepass called Binding Time Analysis (BTA). The results of these decisions are made available to the specializer as annotations on the source program. Because most of the specializer's behavior[3] is determined by the source program and the binding time annotations, both of which are supplied as part of the known (constant) input when the specializer is specialized, much of the specializer's behavior can be determined (and the corresponding reductions performed) at self-application time. In particular, the resulting program generator will contain no code to examine the binding times of any of its inputs or to make reduce/residualize decisions. The residual program will of course contain code to perform reductions and construct residual expressions, but the reduce/residualize pattern will be fixed. In the case of specializing the specializer on an interpreter, the resultant program generator bears a similarity to a compiler, in that it makes reductions based on the "static semantics" (syntactic dispatch, static environment lookup, static typing) [14] of the program and constructs residual code to implement the "dynamic semantics" (dynamic typing, store operations, primitive reductions) of the program. No unnecessary comparisons are performed; the program generator "knows" that the information necessary to reduce the static semantics will be present, and that the information needed to reduce the dynamic semantics will not be present, because that information (the "program division" of [30]) was explicitly available when the program generator was constructed.

There is a tradeoff here between the efficiency of the program generator and the the quality of the programs it produces. The program generator's efficiency depends on making all of the specializer's reduce/residualize decisions reducible at program generator generation time, which is accomplished by "hoisting" them out of the specializer into the BTA prepass. However, this

[2] Of course, we know more about the specializer's inputs than this. Not only do we know that the program to be specialized is `interpreter-1`, we also know that the interpreter will be specialized with its first argument (the program) known and its second arguments (the inputs) unknown. As we shall see in Section 2.1, failure to make use of this additional information will result in an inefficient `specialized-specializer`.

[3] This includes syntactic dispatch, environment lookup, and reduce/residualize choices, but not the actual results of reductions, since the input values for the reductions are determined by the input values to the specializer.

"hoisting" is, by definition, approximate, since some binding time information is unavailable at the time the BTA runs [45]. Thus, specializing an offline specializer yields an efficient generator of potentially inefficient specialized programs; making all reduce/residualize decisions at program generator generation time means that, at program generation time, some performable reductions may not be performed.

Online program specializers are distinguished by their willingness to make reduce/residualize choices at specialization time. This makes them inherently slower than their offline counterparts, due to the need to represent unknown values explicitly, and to make more complex decisions at specialization time (rather than just consulting precomputed annotations). However, because the specializer's reduce/residualize decisions are based on specialization-time information, rather than on BTA-time approximations to specialization-time information as in offline specializers, better choices can be made, resulting in smaller, faster residual programs. This desire for better residual programs has motivated much of the research in offline specialization [45]. If one could construct an efficient program generator by specializing an online specializer, it would not be as fast as a program generator constructed from an offline specializer, because it would still make some of its decisions at program generation time. In the case of specializing the specializer on an interpreter, we would expect the following of the resultant program generator: reductions relating to the static semantics of the program should be performed without examination of the binding times of the program text or interpreter data structures used to evaluate those semantics, since the static nature of those structures should be deducible at program generator generation time, while operations relating to the dynamic semantics may conditionally be reduced *or* residualized, depending on the availability of sufficient information to reduce them. In other words, a program generator constructed from an online specializer is an "optimizing compiler" which is willing to reduce some of the dynamic semantics of the program, rather than an "unoptimizing compiler" which assumes that none of the dynamic semantics will be reducible.

Both offline and online specializers have been successfully specialized, but, to date, only offline specializers and online specializers using offline-style binding time approximation techniques[4] have yielded efficient program generators. Indeed, the conventional wisdom holds that explicit binding time approximations are essential to efficient program generation. This paper demonstrates that, although such methods are indeed effective for efficient program generation, they are not essential. We will demonstrate that the specialization of a nontrivial *online* program specializer without BTA techniques can indeed yield an efficient, accurate program generator. Our solution will require not only a careful choice of specializer data structures, but also a particularly powerful specializer to ensure that the information in those data structures is not prematurely lost (generalized) at program generator generation time. Because of the complexity of such a specializer, we will not demonstrate full self-application; instead, we will show that a nontrivial online program specializer with power at least equivalent to that of MIX [31] and Schism [10] can yield an efficient program generator when specialized by the more powerful specializer FUSE [55, 46]. We will specialize our small online specializer on several programs, and will evaluate the efficiency of the results.

This paper has five sections. Section 1 describes a small online specializer, TINY, which will be used in subsequent examples. In Section 2, we demonstrate a naive approach to program generator generation, the inefficiency of the resultant program generators, and our solution to this problem using FUSE. The third section describes several extensions to TINY, and how they affect the

[4]The "specialization-time BTA" approach of Glück has indeed yielded an efficient, though non-optimizing, program generator. Section 4.2 treats Glück's work in more detail.

problem of program generator generation. Section 4 describes related work in efficient program specialization, using both offline and online techniques. We conclude with several directions for future work in online specialization.

1 TINY: A Small Online Specializer

This section describes TINY, a small but nontrivial online program specializer for a functional subset of Scheme [41] which we will use to demonstrate the construction of online program generators. For reasons of clarity, we will first describe a "watered-down" version of TINY using a denotational-semantics-like language, then describe the actual Scheme implementation. This will allow us to use the abbreviated denotational description in many of the examples in later sections, dropping down into the implementation only when necessary.

The version of TINY we will describe here specializes programs written in a first-order, functional subset of Scheme. A program is expressed as a single `letrec` expression whose body is the name of the goal procedure to be specialized. Both scalars and pairs are supported, but there are no vectors, and no support for partially static structures; that is, any pair containing a dynamic is considered to be dynamic.

1.1 Abstract Description

In this section, we will consider a fragment of TINY which partially evaluates a Scheme expression in an environment mapping Scheme identifiers to specialization-time values, returning a specialization-time value. We are primarily interested in how TINY makes reduce/residualize decisions, so that we can examine whether these decisions can be made at the time TINY is specialized. In a first-order language, the interesting reduce/residualize decisions are at conditionals and primitive applications; we will ignore (for now) how the specializer makes generalization decisions, and how it creates, caches and re-uses specializations of user functions.

For brevity, our description will be couched in a denotational-semantics-like language similar to that used in [6], with double brackets around Scheme syntactic objects. Injection and projection functions for sum domains will be omitted. Of course, the real specializer is written in Scheme and operates on (preprocessed) Scheme programs.

TINY represents each specialization-time values as a *pe-value* (an element of the domain *PE-Value*), as shown in Figure 1. Static values (those known at specialization time) are represented as Scheme values, while dynamic values (those unknown at specialization time) are represented as source language expressions. In the latter case, the expression will compute the runtime value(s) of the specialization-time value. We assume the existence of some helper functions for looking up identifiers in an environment and for coercing values to constant expressions. The fragment of TINY shown in Figure 2 partially evaluates expressions. The function *PE* takes a Scheme expression and an environment mapping each Scheme identifier to a *pe-value*, and returns a *pe-value*.

TINY's online nature can easily be seen in the code for processing `if` expressions. After partially evaluating the test expression, e_1, the specializer tests the resultant *pe-value*, p_1, to see if it is a value (*i.e.,* static). If the value is static, it is used to partially evaluate one of the two arms; otherwise, both arms are partially evaluated and a residual `if` expression is returned. Similarly, the code for processing `car` and `cdr` expressions tests the *pe-values* obtained by partially evaluating the argument expressions. If we construct a program generator by specializing TINY, we would

Domains

e	$\in$	Exp	*expressions (source and residual)*
x	$\in$	Id	*identifiers*
v	$\in$	Val	*scheme denotable values*
p	$\in$	$PEVal = Val + Exp$	*specialization-time values*
env	$\in$	$Env = Var \to PEVal$	*specialization-time environments*

Valuation Functions

PE	:	$Exp \to Env \to PEVal$	*partially evaluate an expression*
$lookup$	:	$Id \to Env \to PEVal$	*look up an identifier*
$resid$	:	$PEVal \to Exp$	*coerce a value to an expression*
car	:	$Val \to Val$	*primitive*
$cons$	:	$Val \to Val \to Val$	*primitive*
$\vdots$			

Figure 1: Domains and Valuation Functions for TINY

$$
\begin{array}{ll}
PE[\![\texttt{(quote } v\texttt{)}]\!]\, env & = v \\
PE[\![\texttt{x}]\!]\, env & = lookup[\![\texttt{x}]\!]\, env \\
PE[\![\texttt{(if } e_1\ e_2\ e_3\texttt{)}]\!]\, env & = let\ p_1 = PE[\![e_1]\!]\, env\ in \\
& \quad\ p_1 \in Val \to \\
& \quad\quad (p_1 = true \to PE[\![e_2]\!]\, env,\ PE[\![e_3]\!]\, env), \\
& \quad\quad let\ p_2 = PE[\![e_2]\!]\, env \\
& \quad\quad\quad\ p_3 = PE[\![e_3]\!]\, env \\
& \quad\quad\quad in\ [\![\texttt{(if } p_1\ (resid\ p_2)\ (resid\ p_3)\texttt{)}]\!] \\
PE[\![\texttt{(car } e_1\texttt{)}]\!]\, env & = let\ p_1 = PE[\![e_1]\!]\, env\ in \\
& \quad\ p_1 \in Val \to (car\ p_1),\ [\![\texttt{(car } p_1\texttt{)}]\!] \\
PE[\![\texttt{(cons } e_1\ e_2\texttt{)}]\!]\, env & = let\ p_1 = PE[\![e_1]\!]\, env \\
& \quad\quad\ p_2 = PE[\![e_2]\!]\, env \\
& \quad\ in\ (p_1 \in Val) \wedge (p_2 \in Val) \to \\
& \quad\quad\quad (cons\ p_1\ p_2), \\
& \quad\quad\quad [\![\texttt{(cons } (resid\ p_2)\ (resid\ p_3)\texttt{)}]\!] \\
\vdots & \\
\end{array}
$$

$$resid\ p = p \in Val \to [\![\texttt{(quote } p\texttt{)}]\!],\ p$$

Figure 2: Fragment of TINY

like to eliminate not only the syntactic dispatch on the first argument of PE, but also as many of these ($p \in Val$) tests as possible. This elimination is the topic of Section 2.

1.2 Implementation Description

The abstract description of TINY omits details relating to concrete data representations, function application, construction and caching of function specializations, and termination. Because these details affect program generator generation, we address them briefly here.

1.2.1 Data Representations

TINY represents source and residual expressions as abstract syntax trees, which are uninteresting relative to our discussion. The choice of a representation for specialization-time values, namely the *pe-values*, is important. The type *PEVal* is a disjoint union of Scheme values and expressions; one obvious way to implement it is with a tagged record, namely either `(static <value>)` or `(dynamic <expression>)`. Many online specializers, such as the one of [7] and the simple online partial evaluation semantics of [15] capitalize on a relationship between expressions and values, namely, that constant expressions of the form `(quote <value>)` are capable of representing values. Thus, they can omit the tag, and can check the "static-ness" of a value merely by checking to see if it is a pair whose `car` is the symbol `quote`. We will see later (*c.f.* Section 2.2.1) that this optimization makes efficient program generator generation far more difficult; thus, TINY does not use it.

(We're still being a bit dishonest here: TINY doesn't actually implement a *pe-value* as a tag followed by a value *or* a residual code expression, it implements it as a tag followed by a value *and* a residual code expression, similar to the *symbolic value* objects of FUSE [54, 55]. This is necessary if partially static values are to be allowed, since the description of the value (*e.g.,* a pair whose `car` is 4 and and whose `cdr` is unknown) may not be deducible from its residual code (*e.g.,* `(cdr (foo x))`). Since this additional mechanism is necessary only for code generation, and not for making reductions, we can, without loss of generality, ignore it for the remainder of the paper. We presented it here only so that the descriptions of partially static encodings in Section 3.2, which also omit code generation details, won't seem strange.)

1.2.2 Function Application

The description in Figure 2 doesn't describe the treatment of user functions. Basically, all that needs to be done is to add an extra parameter representing a set of (*function name*, *function definition*) pairs to the semantics, and to have the semantic function implementing function application look up the function name in this set. After the appropriate formal/actual bindings are added to the environment, the unfolded/specialized body can be constructed via a recursive invocation of the specializer (*i.e.,* a call to the valuation function PE of Figure 2).

1.2.3 Specialization

TINY is a polyvariant program point specializer [8, 30]; that is, it constructs specializations of certain program points (in this case, user function applications) and caches them for potential re-use at other program points (function applications with equivalent argument vectors). TINY builds specializations in a depth-first manner; it does this by adding a single-threaded cache parameter to the semantics, and posting "pending" and "completed" entries to the cache for each specialization before and after it is completed, respectively.[5]. When the specialization process is complete, the cache will contain definitions for the specialization of the goal function, and any specialization it may (transitively) invoke. The code generator uses these definitions to construct the residual program.

1.2.4 Termination

In order to terminate, TINY needs to build a finite number of specializations, each of finite size. The latter can be achieved by limiting the amount of unfolding performed, while the former requires that specializations be constructed only on a finite number of different argument vectors. Most online specializers implement these restrictions using a combination of static annotations (finiteness assertions and argument abstraction) and dynamic reasoning (call stacks, induction detection, argument generalization, and explicit filters). TINY uses static annotations for both types of restrictions. Each user function is tagged with a flag specifying whether it is to be unfolded or specialized, while each formal parameter is tagged with a flag specifying whether it should be abstracted to "dynamic" before specialization is performed. Since much of the power of online specialization derives from its use of online generalization rather than static argument abstraction [45, 46] this might make TINY appear overly simplistic. This is not the case, as adding online termination mechanisms (at least simple ones) does not appreciably increase the difficulty in obtaining an efficient program generator. In Section 3.1, we will show that is is the case.

1.3 Example

In this section, we show two examples of TINY in action. The first example shows the specialization of a very small fragment of a hypothetical interpreter, which is rather small and unrealistic, but will be useful later as an example for program generator generation. Our second example shows the specialization of an interpreter for a small imperative language.

1.3.1 Interpreter fragment example

Consider an interpreter for a small imperative language which maintains a store represented as two parallel lists: `names`, which holds a list of the identifiers in the program being interpreted, and `values`, which holds a list of the values bound to those identifiers. Assume that the interpreter contains an expression of the form `(cons (car names) (car values))`; this might be part of a routine to construct an association list representation of the store to be used as the final output of the interpreter. (The expression `(cons (car names) (car values))` is a standard example; we are following the treatment of [7, 6, 20]. The real purpose of such an expression is irrelevant; all that is important is that the binding times of `names` and `values` differ at the time the interpreter is specialized.)

When the interpreter is specialized on a known program but unknown arguments, the list `names`, which is derived from the program, will be static, but the the list `values`, which is derived from the arguments, will be dynamic. Thus, the specializer will generate a residual constant expression for

[5] For a more formal description of a single-threaded cache, see [15].

```
(lambda (names values) (cons (car names) (car values)))
```

Figure 3: A fragment of a hypothetical interpreter

`(car names)`, and residual primitive operations for `(car values)` and `(cons (car names) (car values))`.

Rather than examining the entire interpreter, we will abstract this small fragment into a separate program (Figure 3). We can use TINY to specialize this program on `names=(a b c)` and `values` unknown by executing the form

```
(tiny fragment-program (list (make-static-peval '(a b c))
                             (make-dynamic-peval)))
```

TINY returns a residual program expressed as a cache; after simple postprocessing (not including dead parameter removal or arity raising), we obtain

```
(lambda (names values) (cons 'a (car values)))
```

which is what we expected. We will return to this example in Section 2, where we will generate a program generator for this fragment by specializing TINY on the fragment and unknown inputs.

1.3.2 MP interpreter example

In this paper, we are not particularly concerned with the efficiency of the specializations constructed by TINY; Instead, we will concern ourselves with the efficiency of the specialization process itself, and the gains to be obtained by program generator generation. However, we would like to show that TINY is indeed a realistic program specializer. To this end, we will show the operation of TINY on an interpreter for a small imperative language.

The MP language [50] is a small imperative language with `if` and `while` control structures, which has become the "canonical" interpreter example for specializers. Figure 4 shows an interpreter for this language; it traverses the MP program's commands and expressions in a straightforward recursive-descent manner, while passing a single-threaded representation of the program's store. Because TINY doesn't handle partially static structures, the interpreter represents the store as two parallel lists, one (static) list for the names, and another (potentially dynamic) list for the values.

When we specialize the MP interpreter on a known program and an unknown input, we expect that all reductions depending solely on the static data (*i.e.,* on the program text) will be performed. Thus, all syntactic dispatch should be eliminated, while store operations should be implemented as open-coded tuple operations. This is exactly what happens; if we specialize an MP program to compare the lengths of two lists:

```
(letrec
    ((mp (lambda (program input)
           (let ((parms (cdr (cadr program)))
                 (vars (cdr (caddr program)))
                 (main-block (cadddr program)))
             (mp-command main-block (init-names parms vars) (init-vals parms vars input)))))

     (init-names (lambda (parms vars)
                   (if (null? parms)
                       (if (null? vars)
                           '()
                           (cons (car vars) (init-names parms (cdr vars))))
                       (cons (car parms) (init-names (cdr parms) vars)))))

     (init-vals (lambda (parms vars input)
                  (if (null? parms)
                      (if (null? vars)
                          '()
                          (cons '() (init-vals parms (cdr vars) input)))
                      (cons (car input) (init-vals (cdr parms) vars (cdr input))))))

     (mp-command (lambda (com names vals)
                   (let ((token (car com)) (rest (cdr com)))
                     (cond
                      ((eq? token ':=)
                       (let ((new-value (mp-exp (cadr rest) names vals)))
                         (update names vals (car rest) new-value)))
                      ((eq? token 'if)
                       (if (mp-exp (car rest) names vals)
                           (mp-command (cadr rest) names vals)
                           (mp-command (caddr rest) names vals)))
                      ((eq? token 'while) (mp-while com names vals))
                      ((eq? token 'begin)
                       (mp-begin rest names vals))))))

     (mp-begin (lambda (coms names vals)
                 (if (null? coms)
                     vals
                     (mp-begin (cdr coms) names (mp-command (car coms) names vals)))))

     (mp-while (lambda (com names vals)
                 (if (mp-exp (cadr com) names vals)
                     (mp-while com names (mp-command (caddr com) names vals))
                     vals)))

     (mp-exp (lambda (exp names vals)
               (if (symbol? exp)
                   (lookup exp names vals)
                   (let ((token (car exp))
                         (rest (cdr exp)))
                     (cond ((eq? token 'quote) (car rest))
                           ((eq? token 'car) (car (mp-exp (car rest) names vals)))
                           ((eq? token 'cdr) (cdr (mp-exp (car rest) names vals)))
                           ((eq? token 'atom) (not (pair? (mp-exp (car rest) names vals))))
                           ((eq? token 'cons)
                            (cons (mp-exp (car rest) names vals)
                                  (mp-exp (cadr rest) names vals)))
                           ((eq? token 'equal)
                            (equal? (mp-exp (car rest) names vals)
                                    (mp-exp (cadr rest) names vals))))))))

     (update (lambda (names vals var val)
               (let ((binding (car names)))
                 (if (eq? binding var)
                     (cons val (cdr vals))
                     (cons (car vals) (update (cdr names) (cdr vals) var val))))))

     (lookup (lambda (var names vals)
               (let ((binding (car names)))
                 (if (eq? binding var)
                     (car vals)
                     (lookup var (cdr names) (cdr vals)))))))
  mp)
```

Figure 4: Interpreter for MP

```
(letrec
 ((mp-while3
   (lambda
    (vals)
    (if
     (caddr vals)
     (mp-while3
      (cond
       ((car vals)
        (if
         (cadr vals)
         (cons (cdar vals) (let ((T44 (cdr vals))) (cons (cdar T44) (cdr T44))))
         (cons
          (car vals)
          (let ((T68 (cdr vals))) (list* (car T68) '() 'a (cdddr T68))))))
       ((cadr vals)
        (cons
         (car vals)
         (let ((T93 (cdr vals))) (list* (car T93) '() 'b (cdddr T93)))))
       (else
        (cons
         (car vals)
         (let ((T119 (cdr vals))) (list* (car T119) '() 'ab (cdddr T119)))))))
     vals)))
  (mp2
   (lambda
    (input)
    (mp-while3
     (let
       ((T22 (list* (car input) (cadr input) '(() ()))))
       (cons
        (car T22)
        (let ((T23 (cdr T22))) (list* (car T23) '#T (cddr T23)))))))))
 mp2)
```

Figure 5: Specialized MP interpreter obtained using TINY

```
(program (pars a b) (dec flag out)
  (begin
    (:= flag '#t)
    (while flag
      (if a
          (if b
              (begin (:= a (cdr a))
                     (:= b (cdr b)))
              (begin (:= out 'a)
                     (:= flag '())))
          (if b
              (begin (:= out 'b) (:= flag '()))
              (begin (:= out 'ab) (:= flag '()))))))))
```

we obtain a specialized interpreter containing only operations pertaining to the values of the identifiers in the program (Figure 5); all syntax and name list comparison operations have been reduced. Under interpreted MIT Scheme, executing the specialized interpreter is approximately 6 times faster than executing the original interpreter on the comparison program. Larger MP programs (or larger inputs to them) produce better speedups. In this paper, we are not particularly concerned with the benefits of specializing interpreters, which are well understood. From now on, we will concentrate only on the efficiency of specializations *of* TINY, rather than on specializations constructed *by* TINY.

2 Program Generator Generation

Having described our simple online partial evaluator, we now turn our attention to producing program generators by specializing it. As we shall soon see (*c.f.* Section 2.2), TINY is insufficiently powerful to produce an efficient program generator when self-applied; constructing an efficient program generator from TINY will require the use of a more powerful specializer. By the end of this section, we will know powerful that specializer must be; for now, we assume the existence of a procedure `specialize`, which takes as arguments the Scheme program to be specialized, and the argument values on which it is to be specialized. To avoid concerning ourselves with this (hypothetical) specializer's representations, we will assume (for now) that, for input purposes, it uses ordinary Scheme values for static values, and `<dynamic>` for dynamic values.

We begin by demonstrating the specialization of TINY on a known program and a completely unknown input specification. We will see that, because this approach fails to specify the binding times of the elements of the input specification, it generates an overly general program generator. The remainder of this section will describe how the binding time information can be represented and maintained, and will give several examples of the generation of efficient program generators from TINY.

2.1 The problem of excessive generality

We will begin by considering the interpreter fragment from Section 1.3. We can accomplish this by specializing TINY on the interpreter fragment and a dynamic[6] argument list, as in

[6]In this instance, "dynamic" means unknown at program generator *generation* time, not at program generation time. That is, the specializer used to specialize TINY knows nothing about the value passed as TINY's second actual

$$
\begin{array}{l}
PE_{\texttt{(cons (car names) (car values))}}\; env = let\; p_1' = let\; p_1 = lookup_{\texttt{names}}\; env\; in \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad p_1 \in Val \rightarrow (car\; p_1),\; [\![\texttt{(car } p_1\texttt{)}]\!] \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad p_2' = let\; p_2 = lookup_{\texttt{values}}\; env\; in \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad p_2 \in Val \rightarrow (car\; p_2),\; [\![\texttt{(car } p_2\texttt{)}]\!] \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad in\; (p_1' \in Val) \wedge (p_2' \in Val) \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (cons\; p_1'\; p_2'), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad let\; p_1'' = p_1' \in Val \rightarrow [\![\texttt{(quote } p_1'\texttt{)}]\!],\; p_1' \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\;\; p_2'' = p_2' \in Val \rightarrow [\![\texttt{(quote } p_2'\texttt{)}]\!],\; p_2' \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad in\; [\![\texttt{(cons } p_1''\; p_2''\texttt{)}]\!]
\end{array}
$$

Figure 6: Fragment of overly general program generator constructed from TINY

```
(specialize tiny-program fragment-program <dynamic>).
```

Consider what happens as TINY is specialized. At best, the specializer can execute all of those operations in TINY's implementation which depend solely on its program input, and on constants in TINY itself. Referring to the description of Figure 2, we can see that the syntax dispatch (all matching of arguments in $[\![\,]\!]$ brackets) can be eliminated. Also, if `specialize` implements partially static structures (or if TINY's implementation maintains the specialization-time environment as two lists, one for the names and one for the values) environment accesses can be reduced to fixed chains of tuple accesses (allowing for other optimizations such as arity raising). However, none of the static/dynamic tests (*i.e.,* those of the form $p_1 \in Val$) or any of the primitive operations (*i.e.,* $(p_1 = true)$, $(car\ p_1)$, or $(cdr\ p_1)$) can be reduced. Of the semantic parameters omitted in Figure 2, the table of function definitions is available, and thus user function lookups can be reduced, but the specialization cache is dynamic, and thus all cache lookups remain residual. An abstract fragment of the resultant program generator is shown in Figure 6. To indicate calls to a specialized version of a semantic function, we subscript the function name with the argument on which it was specialized.

Code of this form is obtained when TINY is self-applied; using a more powerful specializer to specialize TINY on this program does not provide any additional improvement because the necessary binding time information is simply not available.

The program generator of Figure 6 is similar to that obtained by the DIKU researchers [7, 6] when self-applying a simple online specializer on an interpreter. We can view a program generator constructed from an interpreter as a "compiler," since it maps a program written in the language implemented by the interpreter (perhaps a small imperative language; let's call it L) into the language used to implement the interpreter (Scheme).[7] Unfortunately, our program generator isn't a very efficient compiler; it's overly general. In particular, it doesn't know that, at compilation time, the program input will always be static, and the data input will always be dynamic. That is, both

parameter.

[7]Of course, this isn't really the case. Much of the complexity of "real" compilers lies in dealing with resource allocation issues, such as register allocation, memory management, and the like, which have not been addressed by interpreter-based program generation techniques because it's difficult to expose such issues in an interpreter.

```
(program-generator L-program <dynamic>)
```

and

```
(program-generator <dynamic> L-inputs)
```

are perfectly legal invocations of the program generator. The first takes an L-program and returns a Scheme program which maps a list of inputs to the L-program into the output of the L-program; this is what we commonly think of as compilation. The second takes a list of inputs and returns a Scheme program which takes an L-program and applies it to those inputs; this isn't particularly useful since very few (if any) expressions in the interpreter depend on the inputs alone, meaning that the program returned by the program generator is unlikely to be faster than the original interpreter for L. Indeed, our program generator is not a compiler, but instead a specializer specialized on a particular *program* (the interpreter) but not on any particular *argument vector* for that program. Thus, the program generator *must*, by definition, be prepared to accept *any* argument vector, be its elements static or dynamic.

In other words, we got what we asked for; we just asked for the wrong thing. How can we remedy this situation? The answer is that if we want the program generator to have, "built-in," certain assumptions about the binding times of the arguments to the interpreter, we must provide that binding time information to TINY at the time the program generator is constructed. We know of two ways of providing this information:

1. **As binding time annotations on the L-interpreter:** the L-interpreter is augmented with a set of annotations which specify the binding times of each expression in the interpreter; the specializer uses these to make its reduce/residualize decisions. Since the interpreter source (and its binding time annotations) are available when the specializer is specialized, all computations depending solely on the binding time annotations are reduced, and the resultant program generator contains no binding time computations whatsoever. This is the major rationale behind the development of offline specialization techniques [31, 7, 6, 38]. Offline specialization solves the generality problem with relatively little added mechanism in the specializer (indeed, offline specializers are usually smaller than their online counterparts, since specialization-time values no longer need be tagged). Unfortunately, the need to compute the binding time annotations prior to specialization time introduces certain inaccuracies and makes certain optimizations difficult, if not impossible [45].

2. **As part of the arguments on which TINY specializes the L-interpreter:** instead of specifying `<dynamic>` as the type of TINY's second argument (the argument vector to the L-interpreter), specify the binding time of each argument, leaving the values dynamic. That is, instead of executing

   ```
   (specialize tiny-program fragment-program <dynamic>).
   ```

   to construct the program generator, instead use[8]

[8] Note that `make-static-peval` and `make-dynamic-peval` are abstractions specific to TINY, not to `specialize`. We are embedding the meta-value `<dynamic>`, which is an abstraction of `specialize`, in the value slot of a TINY *pe-value*.

```
(define names (make-static-peval <dynamic>))
(define values (make-dynamic-peval))
(specialize tiny-program fragment-program (list names values))
```

which indicates that the `names` argument to `fragment` will be static and the `values` argument will be dynamic at program generation time (when the specialized version of TINY runs). At program generator generation time (when `specialize` runs), we don't know what the static value of the `names` argument to `fragment` *is*; we only know that it's static. Thus, the *pe-value* on which TINY is specialized contains a tag of `static` but a value of `<dynamic>`. In effect, the values on which TINY is symbolically executed by `specialize` mirror those on which TINY was executed in the invocation of TINY on `fragment-program` on page 8, except that the value attribute of the *i.e.,* pe-value representing the static first argument (*i.e.,* `'(a b c)`) has been replaced by an attribute which is dynamic at program generator generation time, but which will be known when the specialized version of TINY runs (*i.e.,* `<dynamic>`).

This solution also produces the desired result, but without the conceptual overhead or accuracy limitations of binding time annotations. It does, however, require significant additional complexity in the specializer used to construct the program generator (our hypothetical `specialize`), relative to TINY or to offline specializers.[9] The remainder of this section describes these additional requirements on the specializer.

2.2 Program generator generation without binding time approximations

The second solution to the problem of an excessively general program generator worked by specifying the binding times of the *pe-values* passed to TINY, but leaving the values of the *pe-values* unspecified. This, in and of itself, is not sufficient to assure that the resultant program generator will contain no unnecessary binding-time-related reductions. There are two issues which must be addressed:

- Representing the binding time information, and
- Preserving the binding time information

We will deal with each of these issues in turn.

2.2.1 Representing binding time information

The essence of our method is that TINY's *pe-value* objects represent both a binding time and a value or residual code expression. By embedding a (specializer specialization time) dynamic value inside a (specialization time) *pe-value*, we can communicate the binding time information attribute of the *pe-value* without being forced to specify the value/expression attribute.

One consequence of this encoding scheme is that the specializer (`specialize`) used to construct the program generator must be able to represent partially static values. That is, it must be able to represent a TINY *pe-value* of the form `(static <dynamic>)` (*i.e.,* a list whose first element is the symbol `static` and whose second element is unknown). Without partially static structures,

[9] If self-application is desired, then these complexity requirements also apply to the specializer being specialized.

`specialize` would represent such a *pe-value* as `<dynamic>`, at which point the binding time information would be lost, and we would once again obtain an overly general program generator. It might at first appear that we could use a binding time separation technique, similar to the parallel name/value lists used to represent stores in interpreters. That is, we could separate the tag and value/expression fields of a *pe-value* into two separate values, so that the dynamic nature of the values at program generator generation time won't pollute the static tags. This is, in effect, what is performed by offline partial evaluation strategies, which separate binding time tags from the input values, attaching them to the program instead.

The problem with such a "separation" approach is that it only works if all of the binding time tags are static; as soon as a single binding time tag becomes `<dynamic>` at program generator generation time under a non-partially-static specializer, the entire binding time environment will be seen as dynamic, and all binding-time-related reductions will be delayed until program generation time. Since some binding times cannot be fully determined until the specialized TINY runs (*e.g.,* values returned out of static conditionals with one dynamic arm, values produced by online generalization, values returned from primitives which perform algebraic optimizations, etc; see [45] for more detailed examples), some of the program generator generation-time representations of *pe-values* will indeed have dynamic binding time tags. Glück's "online BTA" strategy [20] avoids this problem by calculating binding time tags only from other binding time tags. Since all such calculations are, by definition, static, and thus performable at program generator generation time, the resultant program generator will contain no residual binding time reductions. The cost, however, is the same as that of offline BTA; binding times calculated using only knowledge of other binding times (and not of specialization-time values) are necessarily conservative and inaccurate. Program generators constructed in this manner will be unable to make certain optimizations because they will incorporate overly general binding time assumptions.

Thus, we see that, in order build efficient program generators from true online specializers like TINY, the "outer" specializer `specialize` must handle partially static structures.

Another representation problem has to do with TINY's choice of a specialization-time representation for runtime values, *pe-values.* At program generator generation time, we distinguish four different categories of *pe-values*:

1. Known binding time, known value/expression,
2. Known binding time, unknown value/expression,
3. Unknown binding time, known value/expression, and
4. Unknown binding time, unknown value/expression

Choices (1) and (4) are both easy to implement, since in (1), the *pe-value* is completely static, and can be easily represented, while in (4), the *pe-value* can be represented by `<dynamic>`. Choice (3) is unrealistic, since (at least in the non-partially-static version of TINY) the value/expression is sufficient to allow the binding time to be deduced. The problem, then, is how to represent *pe-values* with known binding times but unknown value/expression fields.

Recall that, in Section 1.2.1, we noted that a *pe-value* is a disjoint union of a value and expression, which has several possible representations. TINY uses a representation which separates the union tag from the value/expression field; this allows us to provide a static tag and a dynamic value/expression. An alternate representation used in some specializers optimizes space usage by using constant expressions of the form (quote <value>) to denote static values. Thus, a static *pe-value* is denoted by a quote expression, while a dynamic *pe-value* is denoted by a variable, if, let, or call expression. Unfortunately, this allows us to denote, at program generator generation time, a static *pe-value* with a dynamic value (*i.e.,* (quote <dynamic>) but does not allow us to denote a dynamic *pe-value* with an unknown residual code expression, because the dynamic binding time cannot be distinguished from the expression. To handle this encoding, the outer specializer specialize would have to be able to denote either negations (*i.e., not* (quote <value>)) or disjoint unions (*i.e.,* <dynamic-symbol> *or* (if . <dynamic>) *or* (let . <dynamic>) *or* (call . <dynamic>)); such technology is not presently available. Thus, we will use the (<tag> <value/expression>) encoding of TINY, which requires only that specialize handle partially static structures.

2.2.2 Preserving binding time information

The representational details described in the previous subsection, namely the use of a (<tag> <value/expression>) encoding for TINY *pe-values*, and the handling of partially static structures in specialize, are sufficient to generate efficient program generators for many programs, including the interpreter fragment of Section 1.3.1. However, without additional mechanisms in specialize, some programs will still lead to inefficient program generators. The problem arises when specialize builds a residual loop; if the usual strategy of returning <dynamic> out of all calls to residual procedures is used, binding time information will be lost at that point. Consider the append program:

```
(define (append a b)
  (if (null? a)
      b
      (cons (car a) (append (cdr a) b))))
```

Suppose that we specialize TINY on this program, with a known to be static (but with unknown value) and b known to be a particular static value. During program generator generation, TINY will attempt to unfold the append procedure repeatedly on its static first argument. Since the value of that argument is unknown, specialize will build a specialized version of TINY's unfolding procedure, specialized on the append procedure and an environment where a and b are bound to values with static binding times. This specialized unfolder will contain a recursive call to itself; which, for the program generator to be efficient, must be shown to return a static value. Otherwise, the program generator will contain code for the case where the return value is dynamic, even though it will always be static. Figure 7 shows examples of program generators for append obtained with and without return value reasoning in specialize. Note that this is not a problem for offline specializers because (an approximation to) the binding time of append's return value is computed prior to specialization; the reduce/residualize decision for cons is made by consulting this binding time annotation rather than the return value of the specialized unfolder.

Of course, it is unlikely that anyone would choose to build a program generator for append. However, similar recursive procedures appear frequently in realistic programs like interpreters. For example, the procedure init-names in the MP interpreter of Figure 4 constructs the list of identifiers in the store by recursively traversing a portion of the source program which is static, but with an unknown value, at program generator generation time. This reduces to the same problem as the append example above.

$$
\begin{aligned}
PE_{\texttt{(append a b)}}\, env = {} & let\ p = lookup_{\texttt{a}}\, env\ in\\
& (null?\ p) \rightarrow \texttt{(1 2)},\\
& \qquad let\ env' = update_{\texttt{a}}\, env(cdr\ p)\ in\\
& \qquad\ \ let\ p' = PE_{\texttt{(append a b)}}\, env')\ in\\
& \qquad\quad (p' \in Val) \rightarrow (cons\ (car\ p)\ p'\ ),\\
& \qquad\qquad\qquad [\![\texttt{(cons}\ (car\ p)\ p'\texttt{)}]\!]
\end{aligned}
$$

Program generator fragment obtained without return value reasoning

$$
\begin{aligned}
PE_{\texttt{(append a b)}}\, env = {} & let\ p = lookup_{\texttt{a}}\, env\ in\\
& (null?\ p) \rightarrow \texttt{(1 2)},\\
& \qquad let\ env' = update_{\texttt{a}}\, env(cdr\ p)\ in\\
& \qquad\ \ (cons\ (car\ p)\ (PE_{\texttt{(append a b)}}\, env'))
\end{aligned}
$$

Program generator fragment obtained with return value reasoning

Figure 7: Results of specializing TINY on the `append` program with and without return value reasoning in `specialize`. At program generator generation time, `append`'s first argument is known to be static, but with a dynamic value; its second argument is known to be the static list `'(1 2)`.

$$PE_{\texttt{(cons (car names) (car values))}}\, env = let\ p_1 = lookup_{\texttt{names}}\, env\ in \\ p_2 = lookup_{\texttt{values}}\, env\ in \\ in\ [\![\texttt{(cons (quote } (car\ p_1)\texttt{) (car } p_2\texttt{))}]\!]$$

Figure 8: Fragment of an efficient program generator constructed from TINY

```
;;; executes in an environment where T14 is bound to the pe-value for "names"
;;; (cadr T14) returns the value slot of this pe-value
;;; thus (caadr T14) returns the car of "names"
;;; note that no binding time tags are examined
(list '(dynamic ())                                              ; build a dynamic pe-value
      (list 'code-prim-call                                      ; which is a call
            '(code-identifier cons)                              ; to the primop cons
            (cons (list 'code-constant (caadr T14))              ; on a constant (car names)
                  '((code-prim-call (code-identifier car)        ; and a call to car
                                    ((code-identifier values))))))) ; on identifier values
```

Figure 9: The fragment of Figure 8, expressed as a Scheme program

Thus, our hypothetical specializer `specialize` must be able to infer information about static portions of values returned by calls to specialized procedures.[10] When we consider more powerful versions of TINY (*c.f.* Section 3), we will need corresponding improvements in the information preservation mechanisms of `specialize`.

2.2.3 Example

Thus, we have seen that `specialize` must handle partially static structures and must be able to infer information about static portions of values returned by calls to specialized procedures. The online specializer FUSE [55, 46] meets both of these requirements, so we can use it to specialize TINY.

Now that we have the tools we need, we can demonstrate the generation of an efficient program generator for the interpreter fragment of Figure 3, obtained by specializing the online specializer TINY with the online specializer FUSE.[11] Executing the forms

```
(define names (make-static-peval <dynamic>)
(define values (make-dynamic-peval))
(specialize tiny-program fragment-program (list names values))
```

[10]If TINY were written in a truly tail-recursive style, such as continuation-passing style, `specialize` would not have to reason about return values, but would instead have to reason about static subparts of arguments to continuations with multiple call sites, which is a problem of similar difficulty. See [46, 44] for examples.

[11]What's important here is that TINY is online; it doesn't matter whether the specializer used to specialize it is online or offline, as long as it is sufficiently powerful. However, we do believe that the criteria for `specialize` are easier to achieve using online techniques.

returns an efficient program generator which has, "built-in," not only the syntactic dispatch and static environment lookup, but also the binding times of `names` and `values`. An abstract version of a fragment of this program generator is shown in Figure 8; the corresponding Scheme code from the actual program generator is shown in Figure 9. The efficiency of this program generator is comparable to that of one produced by specializing an offline specializer on the same fragment [7, 6], with the exception of an additional tagging operation to inject the returned residual code fragment into a dynamic *pe-value*.

2.3 Examples

In this section, we give several examples of program generators constructed by specializing TINY on inputs with known binding times, and analyze their performance relative to "naive" program generators constructed on inputs with unknown binding times. The text of these program generators is large and fairly uninteresting, so we will instead describe the program generators in terms their sizes, speeds, and the number of binding time comparisons they perform.

2.3.1 The tests

We tested our program generator generation method on three programs: the `append` program, a regular expression matcher, and the MP interpreter. We constructed a program generator for each of these programs by using FUSE to specialize TINY on that program, and on the binding time values of its inputs. We ran the resultant program generators on one or more actual inputs, and compared the runtime with that of running TINY on the programs and their inputs directly. The example suites were:

- **append**: The program generator was constructed by specializing TINY on the append program, a static[12] first input, and a dynamic second input. The program generator was executed on two inputs:
  - **append(1)**: first argument = `'()`
  - **append(2)**: first argument = `'(1 2 3 4 5 6)`
- **matcher**: The program generator was constructed by specializing TINY on the regular expression matcher program, a static pattern, and a dynamic input stream. The program generator was executed on one input:
  - **matcher**: pattern = $a(b+c)^{*}d$
- **interpreter**: The program generator was constructed by specializing TINY on the MP interpreter, a static program and a dynamic input. The program generator was executed on two inputs:
  - **interpreter(1)**: program = comparison program (*c.f.* Page 11)
  - **interpreter(2)**: program = exponentiation program (*c.f.* [6])

[12] We mean, known to be static, but with value unknown until program generation time.

2.4 Results

Before we continue, we should note that the specialized programs obtained by direct specialization, execution of an efficient program generator, and execution of a naive program generator were identical, modulo renaming of identifiers. This renaming arises because TINY uses a side-effecting operation, `gensym`, to create identifiers, and FUSE does not guarantee that the effect of such operations will be identical in the source program (TINY) and the residual program (the program generators). If TINY were purely functional, the specialized programs obtained by all three means would be, by definition, identical.

The program generators produced for the `append` and matcher examples contained no residual binding time tests outside of the cache lookup routine, which must compare binding time tags because it is comparing tagged values.[13] The program generator produced for the MP interpreter does contain binding time checks for operations depending on values in the store (in Figure 4, all code depending on the formal parameter `vals`) because it cannot be shown at program generation time that this value is dynamic. If the MP program being interpreted doesn't declare any input variables (*i.e.,* it computes a constant value), then its store will be static even if its input is dynamic. Thus, the generated program generator is willing to make reductions based on known values in the store even though the entire store might not be static.[14] These tests can be eliminated by manually inserting generalization operations, but their elimination does not significantly alter the performance of the program generator.

A comparison of the speed of the specializer and our program generators is shown in Figure 10. The speedup ratios, ranging from 3.5 to 20.9,[15] are competitive with those reported by other work on program generator generation [31, 6, 38].

These figures describe benefits due to the use of a program generator, but do not indicate how much of this benefit is due to the use of an *efficient* program generator. To determine this, we constructed naive program generators from TINY by specializing it on completely (program generator generation time) dynamic arguments, and compared the performance and size of the resultant program generators with the efficient program generators constructed above. The results (Figure 11) show that the naive program generators are slower than their efficient counterparts, but are still significantly faster than direct specialization. Of the speedup obtained over direct specialization, 0-46% is traceable to the incorporation of binding time information (and the elimination of binding time reductions). The size ratios are more striking: the naive program generators are 2-37 times larger than the efficient program generators. These numbers are larger than those reported in [6], presumably because Similix factors out primitives into abstract data types (which results in oper-

[13]In a polyvariant online specializer, the cache entries for different specializations of the same procedure may have different binding time signatures; thus, the cache lookup code must compare those signatures, which are not available until program generation time (since the cache contents are dynamic at program generator generation time). This tagging problem does not occur in offline specializers, even those with polyvariant BTA, because all polyvariance with respect to binding time signatures has been expressed via duplication at BTA time. When specializing any particular call site, the specializer (program generator) need only consult a cache, all of whose keys have binding time signatures *known* to be equivalent to the signature of the arguments at the call site. Thus, no tag checks are required.

[14]This makes a lot more sense if the specializer handles partially static structures; TINY will only be able to perform such "optimization" reductions when the entire store is static.

[15]This wide variance can be accounted for by noting that only some portion of the program generator's runtime depends on its inputs; for small inputs, the overhead of cache manipulations, etc, which cannot be optimized to the same degree as syntactic dispatch, will dominate. For example, as the static input to the program generator for `append` increases in length, the amount of time spent in the (highly optimized) unfolding procedure increases.

program	time to specialize using TINY	time to specialize using program generator	speedup
append(1)	14	4.0	3.5
append(2)	143	7.3	19.6
matcher	401	71.0	5.6
interpreter(1)	2449	119.0	20.6
interpreter(2)	4346	208.0	20.9

Figure 10: Speedups due to program generator generation, for various examples. All times are given in msec, and were obtained under interpreted MIT Scheme 7.1.3 on a NeXT workstation. The times given are the average over 10 runs, and are elapsed times (no garbage collection took place). The corresponding times for compiled MIT Scheme are 5-35 times faster, with somewhat lower (approx. 30% lower) speedup figures, presumably due to constant folding, inlining, and other partial evaluation optimizations present in the compiler. Timings include only specialization, not pre- or postprocessing.

program	time (naive)	time (efficient)	speedup
append(1)	4.1	4.0	1.0
append(2)	13.3	7.3	1.8
matcher	98.3	70.8	1.4

Comparison of execution times or naive and efficient program generators

program	size (naive)	size (efficient)	size ratio
append	1071	428	2.5
matcher	32983	874	37.7

Comparison of sizes of naive and efficient program generators.

Figure 11: Comparison of execution times and sizes of naive and efficient program generators. We were unable to construct a naive program generator for the MP interpreter within a 32MB heap; thus, no data are provided for this case. Times are in msec, while sizes are in conses.

ations like `peval-car` or `car` in the program generator), while FUSE beta-substitutes the entire bodies of TINY's primitives. The naive program generators also took correspondingly longer to generate: specialization of TINY with FUSE took 1.2-5.9 times longer, and code generation up to 81 times longer (due to inefficiencies in the current implementation of the FUSE code generator). Large program generators can cause other problems with the underlying virtual machine; *e.g.,* we were unable to compile the naive program generator for the matcher, which contained a 7000-line procedure, in a 32MB heap.

Thus, we see the benefits of restricting the generality of program generators by providing binding time information at program generator generation time.

3 Extensions

The program specializer TINY used in the experiments of Sections 2 and 2.3 is very simple. In this section, we describe several classes of extensions to TINY, and how they affect the difficulty of producing efficient program generators by specializing TINY. We begin (Section 3.1) by adding online generalization, then partially static structures (Section 3.2). Section 3.3 treats two other mechanisms, induction detection and fixpoint iteration, used in online specializers, while Section 3.4 summarizes the difficulties in specializing online specializers, and what is needed to solve them.

3.1 Online Generalization

As described in Section 1, TINY uses an offline strategy for limiting unfolding and for limiting the number of specializations constructed: procedures are explicitly annotated as unfoldable or specializable, and parameters are explicitly annotated as whether they should be abstracted to "dynamic" before specialization is performed.

In [45], we argued that one of the main strengths of online program specializers is their ability to perform *generalization* operations online. Specializers with such a mechanism discard information only when necessary to achieve termination, retaining static values which are common to multiple call sites sharing the same specialization. Thus, it would be desirable if such mechanisms did not have adverse effects on the generation of program generators.

We extended TINY to perform online generalization as follows. Each formal parameter of each user function definition is annotated according to whether it is guaranteed to assume only a finite number of values at specialization time (such annotations can be computed offline, as in [26]). The specializer maintains a stack of active procedure invocations; if it detects a recursive call with identical finite arguments, it builds a specialization on the generalization of the argument vectors of the initial and recursive calls; otherwise, it unfolds the call. For efficiency's sake (*i.e.,* to reduce the size of the stack, and the cost of traversing it at specialization time) we also add an annotation to calls which will always be unfoldable (this can also be computed statically; any nonrecursive procedures, or procedures whose recursion is controlled by finite parameters, can always be unfolded). Because TINY doesn't implement partially static structures, the output of generalization is almost always "dynamic," so there is less to be gained by online generalization than in partially static/higher-order specializers like FUSE; the point here was to determine if this stack mechanism adversely effected the specialization of TINY. Thus, the specializations produced by the version of TINY with online generalization are not appreciably faster than those produced by the version using offline abstraction techniques.

When we specialized the enhanced TINY on the MP interpreter, we still obtained an efficient program generator. Unlike the program generators constructed from the original TINY, this program generator contains specialized unfolding *and* specialized specialization procedures for some of the procedures in the MP interpreter, namely those not annotated as unfoldable. The specialized specialization procedures incorporate all binding time operations on parameters marked as finite (*i.e.,* the program generator performs no tests which will always take only one branch at program generation time) but do contain residual binding time tests for parameters computed via generalization (because the binding times of these parameters are not known until program generation time). This is exactly what we want: statically determinable (*i.e.,* determinable at program generator generation time) operations have been incorporated into the program generator, while operations which cannot be (accurately) performed until program generation time (such as generalization and operations depending on the outputs of generalization) are performed by the program generator.

The version of TINY with online generalization is slower than the version of TINY with static generalization annotations: in the case of the MP interpreter, specialization took 1.9-2.2 times longer, depending on the program being specialized. This ratio carried over into the program generators; the program generator with online generalization was 1.8-2.1 times slower. The speedup due to the use of a program generator instead of direct specialization remained almost unchanged when online generalization was introduced.

Thus, we do not believe that online generalization is an obstacle to the generation of efficient program generators, with some caveats. The outputs of the generalization routine are, naturally, unknown at program generator generation time, so both the generalizer and operations depending on the output of the generalizer are left residual in the program generator. The key to constructing an efficient program specializer with online generalization lies in determining, at program generator generation time, which parameters are liable to be generalized and which aren't. TINY accomplishes this using static finiteness annotations: anything promised to be finite won't be generalized. Without such a static scheme, the outer specializer `specialize` would have to implement complex mechanisms such as equality constraints in order to prove that certain parameters to the generalizer would be guaranteed to be equal at program generation time, allowing the incorporation of the parameters' binding times into the program generator. Thus, we see that offline methods are useful even in the case of online specialization, particularly with respect to termination. This stands to reason—we are not arguing that all operations are best performed online, merely that performing some operations (such as binding time and generalization operations) online has benefits [45] and does not adversely affect the efficiency of program generation.

3.2 Partially Static Structures

TINY does not implement partially static structures; that is, a pair containing one static value and one dynamic value at specialization time is treated as a completely dynamic value. In some cases, partially static structures can be "teased apart" into static and dynamic components either manually or automatically [38, 16], but this is not always possible. Thus, TINY's lack of partially static structures is a limitation.

As we shall see, adding partially static structures to TINY while retaining the ability to produce efficient program generators strains the limits of current specialization technology. We will describe two possible encodings of partially static structures in TINY, and how they affect the specialization of TINY.

```
(define (init-store names values)
  (if (null? names)
      '()
      (cons (cons (car names) (car values))
            (init-store (cdr names) (cdr values)))))
```

Figure 12: Code to initialize a store represented as an association list

3.2.1 Two-tag encoding

Our first encoding scheme retains the structure of the existing TINY encoding, but changes its interpretation. *Pe-values* are still represented as tagged values with the tags static and dynamic, but the value slot of a static *pe-value* must either be an atomic Scheme value or a Scheme pair containing two *pe-values* (instead of two Scheme values, as before). The encoding of dynamic values is unchanged. Thus, a partially static value is simply a static pair containing a dynamic element. No information about completely static values is maintained, as it is not useful in performing reductions (the code generator will find completely static subtrees and generate quote expressions for them, instead of chains of cons primitives). The primitive application, cache lookup, and finiteness annotation mechanisms are changed to accommodate this new representation, but, overall, TINY isn't changed much.

This small change to TINY makes the generation of efficient program generators tremendously difficult; to maintain binding time information encoded in this form, the outer specializer, specialize, must be able to infer and maintain information about disjoint unions and recursive data types. Consider the function init-store (Figure 12), which takes a list of names and a list of values and constructs an association list mapping each name to the corresponding values. Assume that we wish to construct a program generator for an interpreter containing this function, where names is derived from the interpreter's program input, which is known to be static at program generator generation time, while values is derived from interpreter's data input, which is known to be dynamic at program generator generation time. We would expect the program generator to contain a residual loop to construct the store initialization code, and we would expect that the residual loop would be known to return a list of unknown length, but where each element was known to be a pair whose car is known to be static, and whose cdr is known to be dynamic. This would allow later operations involving the names in the store to be reduced (*i.e.,* the program generator contains specialized code to perform these reductions), while operations involving the values would be residualized (without a prior examination of their binding times).

Thus, at program generator generation time, specialize must be able to represent the following types:[16]

[16] We will use identifiers, parentheses, and periods to denote specialize's representations of Scheme objects, while angle brackets and alternate constructions of the form [<a> | <b>] will denote meta-objects of specialize. Thus, <t> ::= [() | ((1 . 2) . <t>)] denotes a list of unknown length consisting of pairs whose car is 1 and whose cdr is 2. The special meta-objects <dynamic> and <atom> denote values not known at the time specialize runs; in addition, <atom> is constrained to denote only atomic Scheme values.

- A completely static *pe-value*; that is, the tag is static and if the value is a pair, both the car and cdr are also completely static *pe-values*.

```
<t1> ::= (static [() | (<t1> .  <t1>)])
```

- A *pe-value* with tag static and whose value is either the empty list or a pair. If the value is a pair, its car is a *pe-value* with tag static and a value which is a pair of a static *pe-value* with unknown atomic value, and a dynamic *pe-value*, and the recursive type.

```
<t2> ::= (static [() |
                    ((static ((static <atom>) .  (dynamic <dynamic>))) .  <t2>)])
```

- A *pe-value* with tag dynamic.

```
<t3> ::= (dynamic <dynamic>)
```

The third type is simple, but the first two are rather complex. Indeed, we know of no program specializer capable of inferring (or even representing) such types; even FUSE only maintains information about types whose size[17] is known at specialization time, reverting to the type <dynamic> for any specialization-time value which might denote arbitrarily large values at runtime. Inferring recursive types is very difficult, because specialize must choose when to collapse a chain of disjoint unions into a recursive type, and must be able to compare and generalize such types. This is difficult because the "collapsing" operation can be done in different ways, with different results; such difficulties are standard when analyzing pointer data structures [9, 24]. Existing approaches to this problem in the pointer analysis community have used either k-bounded approximations, which limit the size of nonrecursive type descriptors, or methods based on limiting each program expression to returning a single type descriptor. This latter approach is also used by the partially static binding time analyses of Mogensen (one binding time grammar production per program point in [38]) and Consel (one type descriptor per *cons point* in [10]), and in the monovariant type evaluator of [58].

Such approaches work well for analyzing an existing program, because the identity of the expressions in the program can be used as "anchor points" to perform least upper bounding and build recursions (as is done in [29, 37, 11]). In the case of an online specializer, which must infer the type of residual code as it is being constructed, we lose this ability to use the identity of code expressions; during the iterative type analysis process, several residual code expressions may correspond to a single program point in the source program. To achieve termination (*i.e.,* to avoid infinite disjoint unions) some of these code expressions (and their types) must be collapsed together, but we can't just collapse together all instances of a source program point as this would yield a purely monovariant specialization. The problem, then, lies in deciding when and how to collapse, or generalize.

Consel's polyvariant partially static BTA [10] operates by keeping all nonrecursive invocations of a procedure distinct, but collapsing recursive call sites together with initial call sites; this works fine for BTA, but will not work for online specialization, as it precludes unfolding of recursive procedures. Aiken and Murphy's type analyzer [1, 39] appears to use a similar method. Mogensen's

[17] By "size," we mean "number of cons cells contained." Even numeric types can become arbitrarily large at runtime, but they are still scalars; there is no need for recursive type descriptors to describe bignums.

higher-order partially static polyvariant binding time analysis [38] is for a typed language, and uses (user- or inferencer-provided) declarations when deciding what recursive types to construct.

One promising approach is the two-stage partial evaluation framework of Katz [34], in which a polyvariant analysis phase computes type information which is then utilized by a separate code generation phase. This offers the possibility that the polyvariant analysis phase could make use of the identity of source program expressions (possibly tagged with some form of instance counter) for building recursive types, without being confused by the construction of multiple residual instances of single program points due to fixpoint iteration and unfolding.

Thus, given the current state of the art, we are unable to construct the first two types listed above at program generator generation time. The consequences of this are disastrous, as we are able to accurately represent only dynamic *pe-values*, and static *pe-values* of known size. All static or partially static *pe-values* of unknown (at program generator generation time) size end up being represented as `<dynamic>`, which contains no binding time information whatsoever. In the case of the interpreter fragment of Figure 12, all binding time information about the parameter `names` and about the value returned by `init-store` is lost. Thus, the resultant program generator will not know that the list of names is static, or that store lookup can be performed statically—indeed, even syntactic dispatch will perform needless binding time comparisons. The program generator will be almost as slow (and large) as a naively generated program generator.

This bodes ill for the self-application of specializers like FUSE, which uses a *symbolic value* encoding similar to the two-tag encoding described in this section. Successful specialization of such specializers appears to require either a change of encoding, or new and more powerful specialization techniques.

3.2.2 Three-tag encoding

In this section, we consider a different encoding of partially static structures, this time using three tag values. The tags `static` and `dynamic` are interpreted as before: `static` denotes a *completely* static value, and is followed by a Scheme value, while `dynamic` denotes a *completely* dynamic value , and is followed by a residual code expression. We add a new tag, `static-pair`, which is followed by a pair of *pe-values*, rather than Scheme values, denoting a pair whose subcomponents are denoted by the corresponding *pe-values*.[18] Thus, `(static-pair ((static 1) . (dynamic (foo x))))` denotes a pair whose `car` is `1` and whose `cdr` is unknown, but can be constructed by the code `(foo x)`.

Compared with the two-tag encoding of Section 3.2.1, this encoding requires slightly more mechanism in the program specializer (TINY) because the `cons` primitive must consult the binding times of its inputs to decide how to tag its output (instead of just always tagging it `static`). Similarly, the code for comparing argument vectors in the cache (and, in the case of online generalization, the stack) must be able to traverse static and partially static pair structures in parallel. However, we will see that this added cost allows more efficient program generators to be generated using existing technology.

Consider the `init-store` code of Figure 12. At program generator generation time, `specialize` must be able to represent the following types:

[18] Of course, a partially static value must also contain the appropriate residual code fragment for constructing the value at runtime. Since this attribute is immaterial to our discussion, we will ignore it.

- A completely static *pe-value*; that is, the tag is `static`.
  ```
  <t1> ::= (static <dynamic>)
  ```

- A *pe-value* which is either the empty list or a partially static pair whose `car` is a partially static pair with static `car` and dynamic `cdr`, and whose `cdr` is the recursive type.

  ```
  <t2> ::= [(static ()) |
            (static-pair ((static <dynamic>) . (dynamic <dynamic>)) .  <t2>)]
  ```

- A completely dynamic *pe-value*; that is, the tag is `dynamic`.
  ```
  <t3> ::= (dynamic <dynamic>)
  ```

Both the first and third types are easy for `specialize` to represent, as it knows the size of all static parts (*i.e.,* both are tuples of length 2). The second type still requires recursive type inference, which is beyond the current state of the art.

Thus, when TINY is specialized on an interpreter containing a call to `init-store`, the resultant program generator will not contain any binding time comparisons for `names` or `values` (or for any syntactic dispatch operations) but will contain needless binding time comparisons in the store lookup routine, because `specialize` lost the information about the structure of the store (*i.e.,* an association list with static `cars`). All binding time operations for completely static or completely dynamic operations are reduced at program generator generation time, but such operations on partially static structures (or their components) are performed online in the program generator. This is significantly better than the results obtained with the two-tag encoding, which couldn't even optimize out operations on completely static structures, but not as good as could be obtained with a more powerful version of `specialize`. Specializing this version of TINY on the MP interpreter using FUSE yielded speedup figures of 11.4-12.0, depending on the MP program being specialized. Because binding time operations on partially static structures are not performed at program generator generation time, this program generator still performs unnecessary binding time manipulations on the store; removing these manipulations would achieve better speedups.

This is an instance of a common phenomenon in partial evaluation, namely the extreme sensitivity of the specializer to changes in the representations used by the problem being specialized. Indeed, the use of a less efficient representation (such as the three-tag encoding here) can often lead to more efficient specializations if the extra work (in this case, having TINY's `cons` primitive consult the binding time tags of its arguments to determine the tag for its result) is performable at specialization time.

3.3 Other Online Mechanisms

In addition to binding time tests in primitives, and generalization, some specializers perform other operations online, such as induction detection [55] and fixpoint iteration [56, 46]. These operations, not present in TINY but present in some versions of FUSE, significantly complicate the task of producing efficient program generators. In this section, we will briefly describe these mechanisms, and explain why present specialization technology cannot handle them.

3.3.1 Induction Detection

Some online specializers such as FUSE [55] and Mixtus [47, 48] make unfold/specialize decisions automatically during specialization. This is usually a two-step process: (1) a recursive call is detected (using a specialization-time call stack), then (2) the specializer decides whether the recursive call poses a risk of nontermination. If there is no risk, the call is unfolded; otherwise, its arguments are generalized with those of the prior call, and a specialization is constructed on the resulting argument vector.

A variety of mechanisms are used for (2); for example, some versions of FUSE generalize only if there is an an intervening dynamic conditional between the initial and recursive calls, thus executing all non-speculative loops (even infinite ones) at specialization time. Another common mechanism is induction detection, which works as follows. The specializer assigns a partial order to all elements of the domain of argument vectors, and unfolds a recursive call only when the recursive call's argument vector is strictly smaller (in the partial order) than the prior call's argument vector. This detects and unfolds static induction, such as `cdr`-ing down a list of known length, or counting down from some number to zero (this only works if a "natural number" type is provided; otherwise, we wouldn't know that the induction is finite at specialization time). Of course, it misses many cases where the iteration space, though bounded, doesn't map monotonically to the partial order (consider a list-valued argument which shrinks by two pairs, grows by one, shrinks by two, etc). However, without help from the programmer in the form of annotations, we cannot hope to detect all such inductions (thus the use of finiteness annotations in some versions of FUSE).

The use of such an induction technique complicates the task of program generator generation because of the need to decide the domain comparisons at program generator generation time. Consider the procedures `mp-command`, `mp-begin`, and `mp-exp` in the interpreter of Figure 4; the syntactic arguments (`com` and `exp`) always strictly decrease on recursive calls, so the specializer will unfold such calls. However, at program generator generation time, only the static nature of these arguments is known; the values are not. When a simple outer specializer `specialize` evaluates the unfold/residualize decision procedure on `(mp-exp (car rest) names value)`, it knows that `(car rest)` is bound to a value with tag `static`, but it doesn't know that the value is strictly smaller than the prior value, `exp`; it just sees `(static <dynamic>)` as the value for both `pe-values`. This means that the comparison is not decidable at program generator generation time, so specialized procedures for both specialization and unfolding are constructed. This increases the size of the program generator, but does not significantly affect its speed. Avoiding this case requires that `specialize` perform inductive reasoning about dynamic values; it would have to notice that the dynamic value in the recursive *pe-value* is derived from the dynamic value in the initial *pe-value* using a string of `car` and `cdr` operations, which means the new value is smaller. It is possible that offline induction analyses such as that of Sestoft [51] could be adapted for this purpose.

However, using this information to make TINY's unfold/specialize decisions at program generator generation time could be difficult; just because `specialize` knows that one list is shorter than another doesn't mean that it can fully evaluate TINY's decision procedure. Because the absolute lengths of the lists are unknown (only relative information is available), the body of the comparison loop (and the `null?` tests contained in the body) cannot be unfolded. Instead, `specialize` must perform theorem proving, using the relative length information to show that, no matter how many iterations the comparison loop performs, its result will always be the same (this can be done by propagating information from the tests of dynamic conditionals (`null?` tests) into the arms, which is not performed by most existing specializers). A simpler solution relies on explicit reflection [52] in the form of an upcall (*i.e.,* instead of explicitly implementing the "shorter" predicate on lists in TINY, make it a primitive in `specialize`, which simply checks to see if one of the lists is derived from the other). However, this would be a less versatile technique; we believe that the results of any reasoning in a specializer such as `specialize` should be usable for improving *any* program, not just special programs containing upcalls.

Some cases are even worse. When `specialize` encounters the decision procedure on the recursive call to `mp-while` in `mp-while`, it will build specialized unfolding and specialization procedures. The specialization procedure will be invoked on the generalization of the current and prior arguments to `mp-while`. Though we can see that the syntactic argument, `com`, is identical in both calls, `specialize` cannot, and thus builds a specialization routine which doesn't know that the binding time of `com` is static, leading to needless binding time comparisons not only in the specialization routine for `mp-while`, but also in the unfolding/specialization routines for any procedures it invokes, such as `mp-command` and `mp-exp`. In short, all knowledge of the static binding time of syntactic arguments is lost, and the program generator is as inefficient (slow and large) as a naively generated one. Avoiding this requires that `specialize` infer and maintain information about the equality of dynamic values, so that when TINY does an `eq?` or `equal?` check on the (dynamic) Scheme value fields of the two static *pe-values*, `specialize` will return true instead of residualizing the decision. Existing specializers do not perform such reasoning, though there is hope that they eventually will, because any specializer hoping to perform well on imperative programs with pointer structures must have such an equality analysis to handle aliasing.

For now, the construction of efficient program generators from specializers using online generalization mechanisms requires that we have some static means of identifying those arguments which cannot possibly be raised to dynamic via generalization, such as those arguments marked with "finite" annotations in TINY. Offline specializers, which by their very nature are prohibited from using online induction analyses, satisfy this restriction trivially, but are of course unable to benefit from such analyses.

3.3.2 Fixpoint Iteration

Another mechanism by which online specializers gain accuracy advantages over their offline counterparts is via fixpoint iteration analyses such as those described in [46]. For example, FUSE can determine that any number of functional updates to a store represented as an association list will preserve the "shape" of the store—that is, the `cars` will remain unchanged. This is important because it avoids unnecessary searching in programs constructed by specializing interpreters (for a more thorough exposition, see Section 2.1 of [46]).

Unfortunately, such analyses encounter problems similar to those faced by specializers with induction detection and online generalization. The problem is that the inner specializer (TINY) decides the equality of two static specialization-time values (for example, the values of parallel keys in two different association lists) by executing the Scheme procedure `equal?`, which cannot be evaluated at program generator generation time, as only the binding times (and not the values) are available. In fact, the two keys will be equal no matter what values are provided at program generation time; their equality is a property of the interpreter, independent of the program on which it is specialized. Making TINY's mechanism work would require that the outer specializer keep track of equality relations between dynamic values, so that the equality tests used for generalization

- Basic online PE
  - partially static structures in `specialize`
  - return value computations in `specialize`
  - explicit tag values in TINY or disjoint union types in `specialize`
- Online Generalization
  - static indication of "ungeneralizable" values in TINY (*i.e.,* finiteness analysis) or equality reasoning in `specialize`
- Partially Static Structures
  - recursive types in `specialize`
- Induction Detection
  - size reasoning in `specialize`
  - propagation of information from tests of dynamic conditionals into arms in `specialize` or primitives for size predicates used in TINY.
- Fixpoint Iteration
  - equality reasoning in `specialize`

Figure 13: Online features and mechanisms for specializing them

and termination of fixpoint iteration would be decidable at program generator generation time.

This might well be a fruitful area for future research even in the domain of offline specializers, because the static reasoning needed to prove that the shape of a store doesn't change, given only the source text of the interpreter (but not of the program being interpreted) could just as easily be performed at BTA time as at program generator generation time. If such an analysis could be provided, then the fixpoint iterations themselves might become unnecessary—the analysis could prove that the names in the store would remain unchanged across iterations instead of having to rediscover this fact for each different set of names (Of course, if we wish to preserve as much information as possible about the *values* in the store, online generalization is still necessary, because the behavior of the values is determined by the program text, not just the interpreter text. Similarly, an interpreter for a language like BASIC, where the store can potentially grow during a loop due to automatic initialization of undeclared identifiers, requires online methods because the equality of store shapes on recursive calls is not provable given the interpreter text alone).

3.4 Summary

Figure 13 summarizes our discussion of the features of online specializers and the mechanisms needed to produce efficient program generators from specializers with such features. Each feature is listed, along with the necessary mechanisms. Together, FUSE and TINY meet these constraints for the first two features, online specialization and online generalization. By choosing appropriate of representations, we achieved some success for partially static structures. Full efficiency with partially static structures, induction detection, or fixpoint iteration will require new specialization technology.

We should also note that Figure 13 should probably include a line for side effects. Many online mechanisms can be implemented far more efficiently using side effects (indeed, FUSE makes very heavy use of side effects to data structures), but if side effects are used, then `specialize` must be prepared to reason about them. Existing techniques, which basically residualize all side-effecting or side-effect-detecting computations, will not be sufficient if binding times are represented using side-effectable data structures (contrast this with the offline case, where binding time information is retained no matter how the specializer handles side effects).

4 Related Work

This section describes related work on program specialization, with an emphasis on techniques for constructing efficient program generators.

4.1 Offline Specialization

Offline specialization and Binding Time Analysis were developed specifically to solve the problem of generating efficient program generators. The first offline specializer, MIX [31, 32] did not handle partially static structures and used explicit unfolding annotations, but was efficiently self-applicable. Subsequent research has produced increasingly powerful offline specializers, which handle partially static structures [38, 10], higher-order functions [38, 6, 11, 21], global side effects [6], and issues of code duplication and termination [51, 6].

The accuracy of offline specializers has been improved through the development of more accurate binding time analyses, such as the polyvariant BTA [10, 38], and *facet analysis* [15], which allows BTA to make use of known properties of unknown values. Other accuracy improvements have been achieved via program transformation, both manually [6] and automatically [38, 16, 13, 27].

In all of the above, efficiency was realized by self-application of the specializer. The usefulness of explicit binding time computations in realizing self-application is described in [7, 6, 30]. Even in the offline world, efficiency methods other than self-application have been used, including the factoring out of computations depending on binding time information (as opposed to only factoring out the binding time computations themselves) [12] and handwriting a program generator generator [28].

Work continues on all of these fronts; however, we do not believe that any current offline specializer is sufficiently powerful to produce an efficient program generator from TINY.

4.2 Online Specialization

The earliest program specializers [33, 3, 23] used online methods. None were suitable for program generator generation, though handwritten program generator generators such as REDCOMPILE [23] were used. Subsequent work on online specialization has focused primarily on accuracy [49, 22, 55, 45, 46, 43, 34, 48] and applications [4, 5, 2, 57] rather than on efficiency.

A notable exception to this is the work of Glück, whose online specializer, V-Mix [20], has been used to generate efficient program generators via self-application. Glück's work makes the same observation as our Section 2.1, namely, that the problem of excessive generality in program generators can be solved without static binding time annotations by encoding the binding times in the arguments passed to the inner specializer. His "metasystem transition" formulation of self-application is very similar to the formulation we gave in Choice 2 on page 13.

Despite these similarities, however, V-Mix and this work differ appreciably in the encoding and preservation of binding time information. In particular, V-Mix uses a *configuration analysis* (described as a "BTA at specialization time") to make reduce/residualize decisions and to compute static/dynamic *approximations* of function results at specialization time (Bondorf outlines a similar approach in [6], p. 34). We would expect such a system to be self-applicable, since, just as in the offline case, all reduce/residualize decisions are made by a process that refers only to the program text and statically available information (binding times), so there is no danger of losing this information at program generator generation time. Unfortunately, such methods share many of the same drawbacks as purely offline methods [45]. For example, binding time approximations to function results computed using only binding time approximations to parameters are overly general—many functions in a program will be given a dynamic return approximation when they might actually return a static value when unfolded at program generation time. Indeed, V-Mix's inability to compute an "unknown" binding time approximation means that all binding time operations will be resolved at program generator generation time, yielding a program generator with *no* online binding time operations even when such operations are necessary to achieve an accurate result.

Indeed, it would appear that the results of *configuration analysis* could be duplicated by a sufficiently accurate polyvariant binding time analysis. The primary benefit of online specialization in V-Mix appears to be the simplicity with which it achieves polyvariance with respect to binding times, not the accuracy of the (essentially offline) program generators it produces. In all fairness, we should note that Glück's work dealt primarily with multiple self-application, in which several levels of specialization were performed, yielding a curried residual program in which each of the first $n-1$ arguments produced a new residual program which was then applied to the next argument (application to the nth argument computed the final result). For this application, the "online BTA" approach worked well, with far less memory consumption than the fixpoint computations used by the outer specializer `specialize` in our examples.

5 Future Work

In this section, we describe several frontiers for future work in the generation of program generators from online specializers.

5.1 Self Application

Although we have demonstrated the specialization of a nontrivial specializer into an efficient program generator, the two specializers (the specializer, `specialize`, and the specializee, TINY) were not the same. This is important if we wish to speed up the process of program generator generation via specialization (*i.e.,* if we specialize the specializer specializing itself, producing a program generator generator, often called a "compiler compiler" in the literature [19, 18, 31]), or if we wish to perform multiple self-application [20] to achieve several levels of currying. It appears as though self-application is achievable only with specializers at particular levels of complexity, where the specializer is simultaneously sufficiently powerful to specialize itself, while being sufficiently simple to be specialized by itself. In the offline paradigm, where the specializer itself performs fairly simple computations directed by static annotations, self-application can be achieved at a relatively low level of complexity.

Unfortunately, this does not appear to be the case for online specializers. Specializing the rather simple specializer TINY required not only partially static structures, but also fixpoint iteration.[19] Specializing a partial evaluator which implements partially static structures require the inference of recursive data types, while fixpoint iteration appears to require inductive reasoning. That is, we have yet to implement mechanisms sufficient to specialize all of the information preservation mechanisms in FUSE, let alone the mechanisms necessary to specialize these as-yet-unwritten mechanisms. More research is required to locate the level of functionality at which closure is achieved, and to determine how to implement such functionality efficiently.

5.2 Encoding Issues

Online specializers necessarily incur an overhead due to the need to represent both static values (all Scheme datatypes) as well as dynamic values using a single universal datatype. Offline specializers for untyped languages can avoid this need for encoding because they have no need to encode dynamic values, and can thus inherit the representation of static values from the underlying virtual machine. Launchbury's lazy encoding technique for typed languages [36] reduces encoding overhead for completely static values, but appears to be of less use for partially static values, because, under online methods, the entire spine from the root of a partially static structure to each of its dynamic leaves must be fully encoded at the time the structure is created (how else could dynamic data be distinguished from static values?). This is not a problem in offline systems like Launchbury's because there is no need to encode dynamic values—the specializer determines dynamic-ness from binding time annotations, not from values.

This encoding has not been a problem to date; specializers such as FUSE, though slower than their offline counterparts, operate at acceptable speeds. Users have been willing to pay the time cost in exchange for the improved accuracy. However, program generator generation brings three encoding problems to light:

First, the outer specializer is forced to partially evaluate all of the inner specializer's encoding and decoding operations (indeed, this is how binding times are deduced). This has large costs in both space and time due to the quadratic explosion in the size of the representations of values; *i.e.,* if a specializer executes k instructions per instruction in the program being specialized, then program generator generation takes time k^2. Glück [20] also notes such growth. For small k, as in offline specializers, this is not a problem, but for larger k (compared with offline specializers, online specializers may execute 5-50 times as many instructions per simulated instruction) this growth becomes unacceptable (*i.e.,* if specialization is 5 times slower, program generator generation may be 25 times slower). For example, specializing TINY on the MP interpreter with FUSE required 7 minutes and a 32 megabyte heap.

Second, the generated program generator often encodes values unnecessarily; *i.e.,* the program generator inherits the encoding used by the specializer it was generated from, and encodes (tags) values with binding time information even when those binding times are never examined. Of

[19] This stands to reason because most abstract interpretation-based binding time analyses also require fixpoint computations.

course, any value which might reach a binding time test must be tagged, but current *arity raising* [42] techniques are insufficiently powerful to remove all "dead" tags. In particular, current arity raisers eliminate only static portions of *parameter* values, without removing static portions of *returned* values. CPS-converting [53] programs will take care of the problem of returned values, but may require a fairly sophisticated dead-code analysis in addition to any complexities added by the higher-order nature of CPS code. Worse yet, if the specializer is written to return tagged values at top level (*i.e.,* FUSE returns a residual program containing, among other things, encoded values), then the program generator must do the same, reducing the number of "dead" tags. It is likely that tag optimization techniques for dynamically typed languages [25, 40] could provide significant improvements here, not only for the specialization of specializers, but the specialization of interpreters for dynamically typed languages as well.

Finally, as we saw in Sections 1.2.1 and 2.2.1, the need to represent values with known binding times and unknown values at program generator generation times constrains the encoding scheme of the inner specializer. In particular, the need to store the binding time and value in separate tuple slots (as opposed to using a distinguished value such as `quote` to indicate static values), enlarges the encoding of values, leading to higher space consumption both at specialization time and at program generation time.

5.3 Accurate BTA

Another potential direction for research in program generator generation is the use of binding time analysis techniques which do not force the specializer into overly general behavior. That is, when two abstract "static" values are generalized, the result should be "unknown binding time" rather than "dynamic." Only decisions involving expressions with "static" and "dynamic" binding time annotations would be performed at BTA time; decisions involving expressions annotated as "unknown binding time" would be delayed until specialization time, as in online specialization.[20] Consel describes such a binding time lattice in [10] but expresses concerns over the pollution of entire expressions due to a single subexpression having an unknown binding time. Bondorf's Treemix [6] uses such an analysis, but its effectiveness is not described.

The motivation behind such methods is twofold. First, it could eliminate the need for encoding values which only reach expressions with known binding times, such as the program in the interpreter example or the pattern in the regular expression example, both of which are known to be static. Second, program generator generation would be simplified: all program generator generation time knowledge of binding times could be provided through the binding time annotations, eliminating the need for the more complex mechanisms of Section 2.2.

However, this approach does have some difficulties. For a specializer like FUSE, which can represent typed unknown values, the choice of binding time domain may be difficult; it may be possible to adapt the *facet analysis* of [15] for this purpose. A highly polyvariant analysis would be required; otherwise, the binding times of several variants would be collapsed into one, forcing greater numbers of binding-time-related reductions to be delayed until program generation time. The use of a binding time analysis for online specialization is also complicated by the need to approximate the specializer's termination mechanism at BTA time. Unlike offline specializers, which rely on

[20] The difference between this approach and traditional offline specialization lies in the fact that, in offline systems, *all* binding times must be computed at BTA time, and *all* reduce/residualize decisions must be completely dictated by the results of the BTA. Thus, a system with an "accurate BTA" would be an (optimized) *online* specializer.

binding time annotations to perform *abstraction* of arguments (*i.e.,* lift specialization-time-infinite static values such as "counters" to dynamic), online specializers perform *generalization* of pairs of argument values at specialization time [45]. Since the values are unavailable at BTA time, it is not possible to determine if the generalization of two static values will be static, meaning that, under a naive BTA, all static arguments to recursive procedures would be raised to the "unknown binding time" value, losing the two benefits described in the previous paragraph. Finally, there is some question as to the benefits to be gained, since online specializers spend a higher proportion of their effort on operations which cannot be optimized given knowledge of binding times, such as recursion detection, generalization, and other information preservation mechanisms.

5.4 Inefficient Program Generators

The motivation for the use of binding time information at program generator generation time is to increase the efficiency of the resultant program generator by avoiding unnecessary reductions at program generation time. Inefficient program generators, which take advantage of the static values available at program generator generation time (*e.g.,* perform syntactic dispatch and environment lookup operations at program generator generation time) but do not make use of binding time information, are both larger and slower than their efficient counterparts. However, preliminary experiments conducted by the authors suggest that the loss in speed may not be particularly large; even *inefficient* program generators are significantly faster than general specializers when it comes to program generation. This suggests that if we could control the size of inefficient program generators, we could construct acceptably efficient program generators without the difficulties inherent in making use of binding time information.

Conclusion

We have shown that, given a careful encoding of specialization-time values and a sufficiently powerful specializer, we can construct an efficient program generator from a simple yet nontrivial online program specializer. We believe this to be the first published instance of efficient program generator generator from an online program specializer without the use of binding time approximation techniques. This result is significant because it allows for the automatic construction of program generators which make online reduce/residualize decision, enabling, for example, optimizing "compilers." It is also a demonstration of the power of online specialization techniques, since the information preservation mechanisms used to achieve efficient program generation, are, at present, implemented only in an online specializer. Unlike binding time approximations, which address only the specialization of specializers, the information preservation techniques used here and in [46] improve the specialization of many programs, not just the specializer TINY. Finally, our result may be of interest to the logic programming community, where, in contrast to the functional programming community, most program specializers use online methods [17, 48, 35].

Nonetheless, this result is unlikely to lead to the widespread proliferation of online-specializer-based program generators. The most obvious reason is that, although we can successfully specialize small specializers such as TINY, we have not developed methods sufficiently powerful to specialize state-of-the-art specializers such as FUSE [55] (*c.f.* Section 3). This forces the user into a choice between efficiency and accuracy of specialization; given that the primary motivation for using online techniques is accuracy, we expect that most users would prefer the slower, more powerful, and as-yet-unspecializable systems. As we suggested before in [45], we believe that the future of program specialization lies in a mixture of online and offline approaches, in which the additional costs of online specialization are paid only when necessary. We leave this to future research.

Acknowledgements

The authors would like to thank John Lamping and Carolyn Talcott for their comments on drafts of this paper.

References

[1] A. Aiken and B. R. Murphy. Static type inference in a dynamically typed language. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 279–290, Orlando, 1991.

[2] W.-Y. Au, D. Weise, and S. Seligman. Generating compiled simulations using partial evaluation. In *Proceedings of the 28th Design Automation Conference*, pages 205–210. IEEE, June 1991.

[3] L. Beckman et al. A partial evaluator and its use as a programming tool. *Artificial Intelligence*, 7(4):291–357, 1976.

[4] A. Berlin. A compilation strategy for numerical programs based on partial evaluation. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, July 1989. Published as Artificial Intelligence Laboratory Technical Report TR-1144.

[5] A. Berlin and D. Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23(12):25–37, December 1990.

[6] A. Bondorf. *Self-Applicable Partial Evaluation.* PhD thesis, DIKU, University of Copenhagen, Denmark, 1990. Revised version: DIKU Report 90/17.

[7] A. Bondorf, N. Jones, T. Mogensen, and P. Sestoft. Binding time analysis and the taming of self-application. Draft, 18 pages, DIKU, University of Copenhagen, Denmark, August 1988.

[8] M. A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.

[9] D. R. Chase. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, White Plains, New York, June 1990.

[10] C. Consel. *Analyse de programmes, Evaluation partielle et Génération de compilateurs.* PhD thesis, Université de Paris 6, Paris, France, June 1989. 109 pages. (In French).

[11] C. Consel. Binding time analysis for higher order untyped functional languages. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 264–272, Nice, France, 1990.

[12] C. Consel and O. Danvy. From interpreting to compiling binding times. In N. Jones, editor, *Proceedings of the 3rd European Symposium on Programming*, pages 88–105. Springer-Verlag, LNCS 432, 1990.

[13] C. Consel and O. Danvy. For a better support of static data flow. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, (LNCS 523)*, pages 496–519, Cambridge, MA, August 1991. ACM, Springer-Verlag.

[14] C. Consel and O. Danvy. Static and dynamic semantics processing. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida*, pages 14–24. ACM, January 1991.

[15] C. Consel and S. Khoo. Parameterized partial evaluation. In *SIGPLAN '91 Conference on Programming Language Design and Implementation, June 1991, Toronto, Canada. (Sigplan Notices, vol. 26, no. 6, June 1991)*, pages 92–105. ACM, 1991.

[16] A. De Niel, E. Bevers, and K. De Vlaminck. Program bifurcation for a polymorphically typed functional language. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 142–153. ACM, 1991.

[17] A. Ershov, D. Bjørner, Y. Futamura, K. Furukawa, A. Haraldson, and W. Scherlis, editors. *Special Issue: Selected Papers from the Workshop on Partial Evaluation and Mixed Computation, 1987 (New Generation Computing, vol. 6, nos. 2,3)*. OHMSHA Ltd. and Springer-Verlag, 1988.

[18] A. P. Ershov. On the partial computation principle. *Information Processing Letters*, 6(2):38–41, April 1977.

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

[20] R. Glück. Towards multiple self-application. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 309–320. ACM, 1991.

[21] C. Gomard and N. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.

[22] M. A. Guzowski. Towards developing a reflexive partial evaluator for an interesting subset of LISP. Master's thesis, Dept. of Computer Engineering and Science, Case Western Reserve University, Cleveland, Ohio, January 1988.

[23] A. Haraldsson. *A Program Manipulation System Based on Partial Evaluation.* PhD thesis, Linköping University, 1977. Published as Linköping Studies in Science and Technology Dissertation No. 14.

[24] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

[25] F. Henglein. Global tagging optimization by type inference. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming (LISP Pointers vol. 5, no. 1, January-March 1992)*, pages 205–215, June 1992.

[26] C. K. Holst. Finiteness analysis. In *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (LNCS 523)*, pages 473–495. ACM, Springer-Verlag, 1991.

[27] C. K. Holst and J. Hughes. Towards binding time improvement for free. In S. Peyton Jones, G. Hutton, and C. Kehler Holst, editors, *Functional Programming, Glasgow 1990*, pages 83–100. Springer-Verlag, 1991.

[28] C. K. Holst and J. Launchbury. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming, Skye, Scotland*, pages 210–218. Glasgow University, 1991.

[29] N. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *Thirteenth ACM Symposium on Principles of Programming Languages, St. Petersburg, Florida*, pages 296–306. ACM, 1986.

[30] N. D. Jones. Automatic program specialization: A re-examination from basic principles. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 225–282. North-Holland, 1988.

[31] N. D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In *Rewriting Techniques and Applications*, pages 124–140. Springer-Verlag, LNCS 202, 1985.

[32] N. D. Jones, P. Sestoft, and H. Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 1(3/4):9–50, 1988.

[33] K. M. Kahn. A partial evaluator of Lisp programs written in Prolog. In M. V. Caneghem, editor, *First International Logic Programming Conference*, pages 19–25, Marseille, France, 1982.

[34] M. Katz and D. Weise. Towards a new perspective on partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Directed Program Manipulation*, San Francisco, CA, 1992. Proceedings available as YALEU/DCS/RR-909.

[35] A. Lakhotia and L. Sterling. ProMiX: A Prolog partial evaluation system. In L. Sterling, editor, *The Practice of Prolog*, chapter 5, pages 137–179. MIT Press, 1991.

[36] J. Launchbury. Self-applicable partial evaluation without s-expressions. In *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (LNCS 523)*, pages 145–164. ACM, Springer-Verlag, 1991.

[37] T. Mogensen. Partially static structures. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347. North-Holland, 1988.

[38] T. Mogensen. *Binding Time Aspects of Partial Evaluation.* PhD thesis, DIKU, University of Copenhangen, Copenhagen, Denmark, March 1989.

[39] B. R. Murphy. A type inference system for FL. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 1990.

[40] J. Peterson. Untagged data in tagged languages: choosing optimal representations at compile time. In *Proceedings of the 4th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 89–99, 1989.

[41] J. Rees, W. Clinger, et al. Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, 4(3):1–55, 1991.

[42] S. Romanenko. Arity raiser and its use in program specialization. In N. Jones, editor, *ESOP '90. 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990. (Lecture Notes in Computer Science, vol. 432)*, pages 341–360. Springer-Verlag, 1990.

[43] E. Ruf and D. Weise. Avoiding redundant specialization during partial evaluation. Technical Report CSL-TR-92-518, Computer Systems Laboratory, Stanford University, Stanford, CA, 1992.

[44] E. Ruf and D. Weise. Improving the accuracy of higher-order specialization using control flow analysis. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Directed Program Manipulation*, pages 67–74, San Francisco, CA, 1992. Proceedings available as YALEU/DCS/RR-909.

[45] E. Ruf and D. Weise. Opportunities for online partial evaluation. Technical Report CSL-TR-92-516, Computer Systems Laboratory, Stanford University, Stanford, CA, 1992.

[46] E. Ruf and D. Weise. Preserving information during online partial evaluation. Technical Report CSL-TR-92-517, Computer Systems Laboratory, Stanford University, Stanford, CA, 1992.

[47] D. Sahlin. The Mixtus approach to automatic partial evaluation of full Prolog. In S. Debray and M. Hermenegildo, editors, *Logic Programming: Proceedings of the 1990 North American Conference, Austin, Texas, October 1990*, pages 377–398. MIT Press, 1990.

[48] D. Sahlin. *An Automatic Partial Evaluator for Full Prolog.* PhD thesis, Kungliga Tekniska Högskolan, Stockholm, Sweden, March 1991. Report TRITA-TCS-9101, 170 pages.

[49] R. Schooler. Partial evaluation as a means of language extensibility. Master's thesis, MIT, Cambridge, MA, August 1984. Published as MIT/LCS/TR-324.

[50] P. Sestoft. The structure of a self-applicable partial evaluator. In H. Ganzinger and N. Jones, editors, *Programs as Data Objects, Copenhagen, Denmark, 1985. (Lecture Notes in Computer Science, vol. 217)*, pages 236–256. Springer-Verlag, 1986.

[51] 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. North-Holland, 1988.

[52] B. C. Smith. Reflection and semantics in Lisp. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages*, pages 23–35, 1984.

[53] G. L. Steele Jr. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1978.

[54] D. Weise. Graphs as an intermediate representation for partial evaluation. Technical Report CSL-TR-90-421, Computer Systems Laboratory, Stanford University, Stanford, CA, 1990.

[55] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture (LNCS 523)*, pages 165–191, Cambridge, MA, August 1991. ACM, Springer-Verlag.

[56] D. Weise and E. Ruf. Computing types during program specialization. Technical Report CSL-TR-90-441, Computer Systems Laboratory, Stanford University, Stanford, CA, 1990. Updated version available as FUSE-MEMO-90-3-revised.

[57] D. Weise and S. Seligman. Accelerating object-oriented simulation via automatic program specialization. Technical Report CSL-TR-92-519, Computer Systems Laboratory, Stanford University, Stanford, CA, 1992.

[58] J. Young and P. O'Keefe. Experience with a type evaluator. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 573–581. North-Holland, 1988.