J. Functional Programming 3 (3): 251–281, July 1993 © 1993 Cambridge University Press

On the specialization of online program specializers[1]

ERIK RUF
Stanford University

DANIEL WEISE
Microsoft Research Laboratory

Abstract

A common technique for improving the speed of program specialization is to specialize the program specializer itself on the program to be specialized, creating a custom *program generator*. 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 program specializers using *binding-time approximations* 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 explicit approximations. This allows the construction of efficient, optimizing program generators from online program specializers.

Capsule review

The paper deals with the problem of generating an efficient program generator by specializing a specializer with respect to a program. This is usually achieved by applying a specializer to itself. The difficulty with this traditional approach is that a trivial specializer may be unable to 'understand' itself because of its 'simple-mindedness', whereas a sophisticated one may be too sophisticated to be understood by itself.

The main idea of the authors is that a 'simple-minded' specializer may be specialized by a sophisticated one, to produce non-trivial (and efficient!) program generators. The paper formulates some requirements that must be satisfied by the pairs of specializers meant for producing residual program generators, and discusses the experimental results obtained. Of particular interest are the techniques of encoding partial-evaluation values that enable the binding-time information to be represented and preserved during partial evaluation.

[1] This research was supported in part by NSF Contract No. MIP-8902764, and by Advanced Research Projects Agency, Department of Defense, Contract No. N0039-91-K-0138. Erik Ruf was funded by an AT&T Bell Laboratories Ph.D. Scholarship.

1 Introduction

A program specializer optimizes a program with respect to an abstract description of its input, producing an optimized ('specialized') program whose applicability is restricted to inputs denoted by the abstract description. As with any context-dependent optimization, the time cost of specialization must be amortized across repeated executions of the specialized program. The number of executions necessary to repay the cost of specialization depends both upon the degree to which specialization improves the speed of the program, and upon the cost of running the specializer.

This tradeoff has motivated much of the recent research in program specialization. Higher degrees of optimization have been achieved by a variety of means, including binding-time analysis, binding-time improvement, and online analysis techniques. Efficiency of the specialization process has also been addressed in several ways, including performing more operations statically prior to specialization time (Consel and Danvy, 1990), and handwriting a 'specializer generator' (Haraldsson, 1977; Holst and Launchbury, 1991).

However, the most popular means of gaining efficiency, independently discovered by Futamura (1971), Ershov (1977) and Turchin, is based on the observation that the program specializer is often called upon to specialize a single program (e.g. an interpreter) on different constant input values (e.g. programs to be executed by the interpreter). Thus, we can benefit by specializing the specializer with respect to the program to be specialized, producing a custom specializer, or *program generator*, for that particular program. We call the process of specializing the specializer *program generator generation*, or, in cases case where the 'inner' and 'outer' specializers are identical, *self-application.*

Much attention has been devoted to techniques for generating *efficient* program generators; e.g. program generators that do not perform unnecessary operations that could instead have been performed when the program generator was constructed. The earliest program specializers (Beckman *et al.*, 1976; Haraldsson, 1977; Kahn, 1982; Turchin, 1986) used online methods, and were not suitable for efficient program generator generation, though handwritten program generator generators such as REDCOMPILE (Haraldsson, 1977) were used. Offline specializers were invented specifically to solve this problem; MIX (Jones *et al.*, 1988) was the first efficiently self-applicable specializer. More recent work has produced increasingly powerful self-applicable offline specializers; for examples, see (Mogensen, 1989; Consel, 1989; Bondorf, 1990; Gomard and Jones, 1991; Jones *et al.*, 1993). Contemporary work on online specialization (Schooler, 1984; Sahlin, 1991; Weise *et al.*, 1991; Katz and Weise, 1992; Ruf, 1993) has focused primarily on accuracy rather than on efficiency, with the notable exception of Glück's work (Glück, 1991; Glück and Turchin, 1989), on the self-application of online specializers and supercompilers.

In all cases to date, the generation of efficient program generators has required the use of explicit *binding-time approximation* techniques. Such methods simplify the task of specializing the specializer, but at the cost of generating less efficient specialized programs, since the approximation of binding times can cause optimizations to be missed.

This paper addresses the goal of obtaining *efficient* program generators without the use of binding-time approximation techniques. 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 similar to that of the original online MIX (Jones *et al.*, 1985) can yield an efficient program generator when specialized by FUSE (Weise *et al.*, 1991). We will specialize our small online specializer on several programs, and will evaluate the efficiency of the results.

The remainder of this paper consists of four sections. We begin with a description of the small online specializer which we will specialize in later sections. Section 3 demonstrates the problem of excessive generality, prior solutions to the problem, and our solution. Section 4 describes how more complex online mechanisms can be specialized. We conclude with a discussion of future work.

2 A small online specializer

This section describes TINY, a small but nontrivial online program specializer for a first-order functional subset of Scheme (Rees *et al.*, 1991), which we will use to demonstrate the construction of online program generators. 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. TINY is similar in complexity to the 'naive MIX' of (Bondorf *et al.*, 1988) and the V-Mix system of (Glück, 1991).

For reasons of clarity, we will first describe a small fragment of TINY in a denotational-semantics-like language, then informally describe the remainder of the actual implementation. This will allow us to use the abbreviated description in many of the examples in later sections, dropping down into the implementation only when necessary.

2.1 Abstract description

Consider the 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.

TINY approximates sets of runtime values with specialization-time approximations, which we will call *pe-values*, which are elements of the domain *PEVal*, as

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*

Function Signatures

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*

⋮

Fig. 1. Domains and Function Signatures for TINY.

```
PE [[(quote v)]] env      = v
PE [[x]] env              = lookup [[x]] env
PE [[(if e₁ e₂ e₃)]] env  = let p₁ = PE [[e₁]] env in
                              p₁ ∈ Val →
                                (p₁ = true → PE [[e₂]] env, PE [[e₃]] env),
                                let p₂ = PE [[e₂]] env
                                    p₃ = PE [[e₃]] env
                                  in [[(if p₁ (resid p₂) (resid p₃))]]
PE [[(car e₁)]] env       = let p₁ = PE [[e₁]] env in
                              p₁ ∈ Val → (car p₁), [[(car p₁)]]
PE [[(cons e₁ e₂)]] env   = let p₁ = PE [[e₁]] env
                                p₂ = PE [[e₂]] env
                              in (p₁ ∈ Val) ∧ (p₂ ∈ Val) →
                                   (cons p₁ p₂),
                                   [[(cons (resid p₂) (resid p₃))]]
        ⋮
resid p = p ∈ Val → [[(quote p)]], p
```

Fig. 2. Fragment of TINY.

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 helper functions for looking up identifiers in an environment and for coercing values to constant expressions. The fragment of TINY shown in Figure 2 specializes 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 specialize one of the two arms; otherwise, both arms are specialized 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 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.

2.2 Implementation description

The abstract description given above treats only a small fragment of TINY, and 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.

2.2.1 Data representations

TINY represents source and residual expressions as abstract syntax trees, which are uninteresting relative to our discussion. However, the choice of a representation for specialization-time values is important. The type *PEVal* is a disjoint union of Scheme values and expressions, and is implemented as a tagged record, namely either `(static <value>)` or `(dynamic <expression>)`.

2.2.2 Function application

The treatment of user functions is straightforward, and was omitted in Figure 2 merely to avoid cluttering the description. If we treat the program as a global, then the semantic function implementing function application will look up the function name in the program, add appropriate formal/actual bindings are added to the environment, and construct the the unfolded/specialized body via a recursive invocation of the specializer (e.g. a call to *PE*.)

2.2.3 Specializations

TINY constructs and caches specializations in a depth-first manner 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 (for a more formal description of a single-threaded cache, see the Appendix of (Ruf, 1993)). 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. Because caching involves no reduce/residualize decisions, we need not concern ourselves with it directly.

```
(define (fragment-program names values)
  (cons (car names) (car values)))
```

Fig. 3. A fragment of a hypothetical interpreter.

2.2.4 Termination

Reduce/residualize decisions do enter the picture when we consider termination. 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. TINY's termination mechanisms requires no reduce/residualize decisions at specialization time. 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.

Most online specializers use some amount of dynamic reasoning (call stacks, induction detection, argument generalization, and explicit filters) to achieve termination. Since much of the power of online specialization derives from its use of online generalization rather than static argument abstraction, TINY's reliance on static annotations makes 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 4.1, we will show that is the case.

2.3 Example

In this section, we show 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.

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 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 list `values`, which is derived from the arguments, will be dynamic. Thus, the specializer will generate a residual constant expression for `(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

```
(define (spec-fragment-program names values)
  (cons 'a (car values)))
```

which is what we expected. We will return to this example in Section 3, where we will generate a program generator for this fragment by specializing TINY on the fragment and unknown inputs.

In this paper, we are not particularly concerned with the efficiency of the specializations *constructed by* TINY, but rather with the efficiency of specializations *of* TINY. However, we would like to briefly note that TINY is indeed a realistic program specializer. For example, using TINY to specialize an interpreter for the 'canonical' MP imperative language on MP programs yields specialized interpreters comparable to those produced by other specializers, and speedups of 6-20 under interpreted MIT Scheme.

3 Program generator generation

In this section, we examine the problems inherent in specializing an online specializer like TINY, the solutions developed to date, and our solution. We begin by demonstrating the specialization of TINY on a known program and a completely unknown input specification. Because this approach fails to specify the binding times of the elements of the input specification, it generates an overly general program generator. This problem is well-known; we examine the solutions described in (Bondorf *et al.*, 1988; Glück, 1991), both of which are based on modifying the specializer being specialized (in this case TINY) in ways that compromise its accuracy. We then describe our solution, and show its performance on several examples.

As we shall soon see, 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 paper, we will know how 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 that, for input purposes, it uses ordinary Scheme values for static values, and the special symbol `<dynamic>` for dynamic values.

3.1 The problem of excessive generality

Consider constructing a program generator for the the interpreter fragment from Section 2.3. We can accomplish this by specializing TINY on the interpreter fragment

$$\begin{array}{l}
PE_{\texttt{(cons (car names) (car values))}}\ env = let\ p'_1 = let\ p_1 = lookup_{\texttt{names}}\ env\ in \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad p_1 \in Val \rightarrow (car\ p_1),\ [\![\texttt{(car } p_1\texttt{)}]\!] \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\ p'_2 = let\ p_2 = lookup_{\texttt{values}}\ env\ in \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad p_2 \in Val \rightarrow (car\ p_2),\ [\![\texttt{(car } p_2\texttt{)}]\!] \\
\qquad\qquad\qquad\qquad\qquad\qquad in\ (p'_1 \in Val) \wedge (p'_2 \in Val) \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (cons\ p'_1\ p'_2), \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad let\ p'_1 = p'_1 \in Val \rightarrow [\![\texttt{(quote } p'_1\texttt{)}]\!],\ p'_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad p'_2 = p'_2 \in Val \rightarrow [\![\texttt{(quote } p'_2\texttt{)}]\!],\ p'_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad in\ [\![\texttt{(cons } p'_1\ p'_2\texttt{)}]\!]
\end{array}$$

Fig. 4. Fragment of overly general program generator constructed from TINY.

and a dynamic (where 'dynamic' means unknown at program generator *generation* time, not at program generation time) argument list, as in

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

When TINY is specialized, `specialize` 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 complete program 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 4. To indicate calls to a specialized version of a semantic function, we subscript the function name with the argument on which it was specialized.

If we assume that our program fragment is embedded in an interpreter, which will always be specialized on a known program (i.e. static `names`), and an unknown input value (i.e. dynamic `values`), then the program generator we obtained by specializing TINY is overly general, since it tests `names` and `values` to see if they are static or dynamic (the tests $p_1 \in Val$ and $p_2 \in Val$ in Figure 4).

Increasing the power of `specialize` does not help; the information necessary to reduce the static/dynamic tests is simply not available. As was noted by Bondorf (1988) and Glück (1991), the problem lies in the fact that we specialized TINY on a particular *program* (the interpreter fragment), 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.

3.2 Existing methods

Existing systems use one of two methods for providing binding-time information at specialization time. The information is provided either:

1. as binding-time annotations on the program, or
2. as part of the arguments on which TINY specializes the program:

We will treat each of these in turn:

3.2.1 Binding-time information via annotations

This is the approach taken by offline specializers (Jones *et al.*, 1988, 1993), which precompute approximations to the binding times of all expressions in the program, then annotate the program text with these approximations. The specializer (in this case, TINY) uses these annotations, rather than any property of specialization-time values, to make reduce/residualize decisions. Since the program source (and thus its binding-time annotations) is 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. Some systems, such as that of (Consel and Danvy, 1990), go one step further, using the annotations to precompute custom reduction or residualization directives, but the essence of the method is the same.

Thus, 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, given only binding-time abstractions of the specialization-time inputs, can cause expressions potentially returning static values to be annotated as dynamic, resulting in missed reductions. Certain optimizations are made more difficult, if not impossible (Ruf and Weise, 1992a). In any case, we cannot use this approach to generate efficient program specializers from an online specializer such as TINY, since it requires that we recode TINY into an offline form.

3.2.2 Providing binding-time information via argument coding

This approach was first published by Glück (1991), using the 'metasystem transition' formalism of Turchin (1986). It uses `specialize`'s ability to represent partially known values to specify the binding-time portion (but not the value/expression portion) of the argument inputs to TINY. That is, instead of executing

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

to construct the program generator, instead use

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

where `make-static-peval` and `make-dynamic-peval` are abstractions specific to TINY, not to `specialize`. 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` earlier, except that the value attribute of the *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.

Providing binding-time information in this way does not guarantee that the resultant program generator will be efficient; that depends on `specialize`'s ability to optimize away binding-time operations in TINY given the partially specified input *pe-values*. We will describe how this is accomplished in Glück's V-Mix system, and why that solution is insufficient for our purposes.

V-Mix achieves efficiency by ensuring that *all* binding-time operations can be performed statically when V-Mix is specialized. It explicitly approximates binding times using a *configuration analysis* (described as a 'BTA at specialization time'), computing approximate binding times for function results given only the binding times of the arguments, not their values (a similar approach is outlined on p. 34 of Bondorf (1990). Thus, given input *pe-values* encoding known binding times, V-Mix is efficiently 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 (the binding times of the inputs). There is no danger of losing this information at program generator generation time.

However, the statically computed binding times are only approximate. For example, binding-time values for function results computed using only binding-time values of parameters can be 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 the generated program generator will have *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.

Our goal in this work is to retain the benefits of online binding-time computations in the program generator. That is, we do not require that *all* binding-time operations in TINY be reducible when TINY is specialized. Of course, those binding-time operations that can be performed at program generator generation time without introducing approximations should be performed then, but those requiring actual static values should be delayed until the program generator is executed.

3.3 Our solution

Our work builds on Glück's insight that binding-time information can be provided by argument coding, but also addresses the problem of preserving binding-time information without explicit approximation techniques. In this section, we describe the constraints on TINY's encodings and `specialize`'s behavior which make this possible, and give examples of specializing TINY under FUSE. There are two issues which must be addressed:

1. Representing the binding-time information, and
2. Preserving the binding-time information

We will deal with each of these issues in turn.

3.3.1 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, `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 themselves static at program generator generation time. Under a non-partially-static specializer, as soon as a single binding-time tag becomes `<dynamic>` at program generator generation time, 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.), some of the program generator generation-time representations of *pe-values* will indeed have dynamic binding-time tags. Thus, we see that, to 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 2.2.1, 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, such as those of (Bondorf *et al.*, 1988; Glück, 1991), uses 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 use the `(<tag> <value/expression>)` encoding of TINY, which requires only that `specialize` handle partially static structures.

3.3.2 Preserving binding-time information

The representational details described in the previous subsection are sufficient to generate efficient program generators for many programs, including the interpreter fragment of Figure 3. 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:

```
PE(append a b) env = let p = lookupa env in
                       (null? p) → (1 2),
                                  let env' = updatea env (cdr p) in
                                   let p' = PE(append a b) env') in
                                    (p'∈ Val) → (cons (car p) p' ),
                                                ⟦(cons (car p) p')⟧
```

Program generator fragment obtained without return value reasoning

```
PE(append a b) env = let p = lookupa env in
                       (null? p) → (1 2),
                                  let env' = updatea env (cdr p) in
                                   (cons (car p) (PE(append a b) env'))
```

Program generator fragment obtained with return value reasoning

Fig. 5. 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)`.

```
(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 5 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 store-building procedure in the 'standard' MP interpreter 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.

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

Fig. 6. Fragment of an efficient program generator constructed from TINY.

```
;;; executes in an environ. 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                                ; on
          (list 'code-constant (caadr T14))  ; a constant (car names)
          '((code-prim-call                  ; and a call
              (code-identifier car)          ; to car
              ((code-identifier values)))))) ; on identifier values
```

Fig. 7. The fragment of Figure 6, 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.[2] When we consider more powerful versions of TINY (c.f. Section 4), we will need corresponding improvements in the information preservation mechanisms of `specialize`.

3.4 Examples

In this section, we give several examples of program generators constructed by specializing TINY on inputs with known binding times. We begin with the interpreter fragment example of Section 2.3, and demonstrate the specialization of TINY on the program of Figure 3, using FUSE. Executing the forms

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

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 6; the corresponding Scheme code from the actual program generator is shown in Figure 7. The efficiency of this program generator is comparable to that of one produced by specializing an offline specializer on the same fragment (Bondorf *et al.*, 1988; Bondorf, 1990), with the exception of an additional tagging operation to inject the returned residual code fragment into a dynamic *pe-value*.

[2] 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 Ruf (1992b; 1993) for examples.

In the remainder of this section, we consider several more realistic programs and their associated program generators, 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 of their sizes, speeds, and the number of binding-time comparisons they perform.

3.4.1 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 first input known to be static, but with unknown value, and a second input known to be dynamic. 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. p 223 of (Ruf, 1993))
  - **interpreter(2)**: program = exponentiation program (c.f. (Bondorf, 1990))

3.4.2 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.

Table 1. *Speedups due to program generator generation, for various examples*

test case	time to specialize[a] 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

[a] All times are given in msec, and were obtained under interpreted MIT Scheme 7.2 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.

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.[3] The program generator produced for the MP interpreter does contain binding-time checks for operations depending on values in the store 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. (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). 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 1. The speedup ratios, ranging from 3.5 to 20.9, are competitive with those reported by other work on program generator generation (Jones *et al.*, 1985; Mogensen, 1989; Bondorf, 1990). We were unable to construct a naive program generator for the MP interpreter within a 32 MB heap; thus, no data are provided for that case. The wide variance of the speedups can be accounted for by noting that only some portion of the program generator's runtime depends on its inputs; for

[3] 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.

Table 2. *Execution times (in msec) of naive and efficient program generators*

test case	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

Table 3. *Sizes (in conses) of naive and efficient program generators*

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

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.

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 in Table 2 show that the naive program generators are slower than their efficient counterparts, but are still significantly faster than direct specialization. The size ratios (Table 3) 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 (Bondorf, 1990), presumably because Similix factors out primitives into abstract data types (which results in operations 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 32 MB heap.

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

4 Extensions

The program specializer TINY used in the experiments of Section 3.4 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 4.1) by adding online generalization, then partially static structures (Section 4.2). Section 4.3 treats two other mechanisms, induction detection and iterative type analysis, used in online specializers, while Section 4.4 summarizes the difficulties in specializing online specializers, and what is needed to solve them.

4.1 Online generalization

As described in Section 2, 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.

Many online specializers use a different strategy, in which arguments are dynamically abstracted only when necessary to achieve termination, retaining those static values which are common to the initial and recursive entries to a loop. The specializer does this by dynamically detecting recursive procedure invocations which might potentially lead to infinite unfolding, then specializing the procedure on the least upper bound of the argument vectors of the initial and recursive call sites. Such schemes have been called 'generalization' (Weise *et al.*, 1991; Turchin, 1988), 'generalized restart' (Sahlin, 1991), and 'respecialization' (Glück, 1991) in the literature. In previous work (Ruf and Weise, 1992a), we have argued that generalization is one of the main strengths of online techniques; 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 Holst (1991). 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, without any loss of generality). Because TINY doesn't implement partially static structures, the output of generalization is almost always 'dynamic',' rather than some partially known value, 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 affected the specialization of TINY.

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 but do contain residual binding-time tests for parameters computed via generalization. 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 abstraction 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 identifying, at program generator generation time, those arguments which cannot possibly be raised to dynamic via generalization, such as those marked with 'finite' annotations in TINY. Without the static finiteness annotations, the outer specializer `specialize` would have to maintain equality constraints to determine the binding time of any parameter value (since the binding time of the result of a generalization depends on the *equality*, rather than just the binding times, of its inputs). Thus, we see that offline methods can be useful even in the case of online specialization, particularly with respect to termination.

4.2 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. 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.

4.2.1 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

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

Fig. 8. Code to initialize a store represented as an association list.

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 special information is maintained for completely static values, as it is not useful in performing reductions.[4] 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 8), 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` will be static and `values` will be dynamic at program 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 static, and whose `cdr` is dynamic. This would allow later operations involving the names in the store to be reduced (i.e. the program generator will contain 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:[5]

- 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>)])
  ```

[4] Such information is useful at code generation time, e.g. for substituting list constants for completely static trees of `cons` primitives, but this can be recovered from the *pe-values*.

[5] 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 *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 (non-pair) 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. FUSE only maintains information about structured types whose size is known at specialization time, reverting to the type `<dynamic>` for any specialization-time value which might denote arbitrarily large values at runtime.

Computing recursive types is a difficult problem, because of the need to collapse a chain of disjoint unions into a recursive type. 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 works well for analyzing a complete program, because the identity of the expressions in the program can be used as 'anchor points' to perform least upper bounding and build recursions. Examples include the partially static binding-time analyses of Mogensen (1989) (one binding-time grammar production per program point) and Consel (1989) (one type descriptor per *cons point*), and the monovariant type evaluator of Young and O'Keefe (1988). 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.

Polyvariant static analyses do not appear to be useful here either. Consel's polyvariant partially static BTA (Consel, 1989) operates by keeping all nonrecursive invocations of a procedure distinct, and 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 (Aiken and Murphy, 1991) appears to use a similar method. Mogensen's higher-order partially static polyvariant binding-time analysis (Mogensen, 1989) is for a typed language, and uses (user- or inferencer-provided) declarations when deciding what recursive types to construct. The online polyvariant analysis phase proposed by Katz (1992), has promise, but has not yet been implemented.

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 8, 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.

4.2.2 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*.[6] 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 4.2.1, 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 8. At program generator generation time, `specialize` must be able to represent the following types:

- A completely static *pe-value*; that is, the tag is `static`.

```
<t1> ::= (static <dynamic>)
```

[6] 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 *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.

4.3 Other online mechanisms

In addition to binding-time tests in primitives, and generalization, some specializers perform other operations online, such as induction detection (Weise *et al.*, 1991) and iterative type analyses (Ruf and Weise, 1992b; Ruf, 1993). 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.

4.3.1 Induction detection

Some online specializers such as FUSE and Mixtus (Sahlin, 1991) make unfold vs. specialize decisions automatically during specialization. One common mechanism for this purpose relies on detecting inductions, and works as follows. The specializer assigns a well-founded partial ordering to all *pe-values*, and unfolds a recursive call only when the recursive call's argument vector is strictly smaller (in the partial ordering) 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). This is conservative, as it fails to unfold cases where the iteration space, though bounded, doesn't map monotonically to the partial ordering (consider a list-valued argument which shrinks by two pairs, grows by one, shrinks by two, etc). However, it has proven useful in practice.

For example, the expression argument of an interpreter typically shrinks on each recursive call to the evaluation procedure. TINY can test this easily by merely comparing the sizes of the expression arguments on successive invocations of the evaluator. At program generator generation time, however, only the binding time (in this case, static) of the argument *pe-values* is known, but not the values (which are dynamic at program generator generation time), meaning that, without additional analysis on the part of `specialize`, TINY's length comparison will not be decidable when TINY is specialized. This is unfortunate, as the inductive traversal property is true for all possible expressions, and thus should be determinable at program generator generation time.

Making this determination requires that `specialize` reason about the sizes of dynamic values; e.g. proving that the dynamic value in the recursive *pe-value* is derived from the dynamic value in the initial *pe-value* via a series of `car` and `cdr` operations. Using this information to evaluate TINY's decision procedure requires even more reasoning: if TINY's decision procedure compares the length of lists by traversing them in parallel, `specialize` must make an inductive argument to prove that, no matter how many iterations TINY's comparison loop performs its result will always be true. 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 (the supercompiler (Turchin, 1986) can do this in some cases). It is likely that future specializers will have such mechanisms, as they are useful, if not essential, in handling aliasing. A simpler solution relies on explicit reflection in the form of an upcall, where TINY's implementation of the 'shorter' predicate on lists is replaced by a primitive known to `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 of this form.

4.3.2 Iterative type analyses

Another mechanism by which online specializers gain accuracy advantages over their offline counterparts is via iterative type analyses, such as those described in (Ruf and Weise, 1992b; Ruf, 1993). 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.

Unfortunately, such analyses encounter problems similar to those faced by specializers with induction detection and online generalization. The problem is that the type analyses test 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. For some interpreters (such as MP), the equality of the keys is a property of the interpreter, independent of the program on which it is specialized, so we ought to be able to decide this at program generator generation time. Doing so would require that `specialize` keep track of equality relations between dynamic values, so that the equality tests used by the type analysis can be decided when the program generator is constructed.

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.

4.4 Summary

Figure 9 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

- 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`

Fig. 9. Online features and mechanisms for specializing them.

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 9 should probably include a line for side effects. Many online mechanisms can be implemented far more efficiently using side effects (FUSE makes heavy use of side effects, to data structures as well as to variables), 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).

5 Future work

In this section, we briefly 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, we have not demonstrated self-application. Self-application is important if we wish to speed up the process of program generator generation via by constructing a 'compiler compiler' (Ershov, 1978), or if we wish to perform multiple self-application (Glück, 1991) to achieve several levels of currying. It appears as though self-application is achievable only with specializers having 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; to date, each mechanism in TINY has required yet more complexity in `specialize`. More research is required to locate the level of functionality at which closure is achieved, and to determine how to implement such functionality efficiently. Doing so may or may not be worthwhile, depending on whether the additional analyses are also useful for specializing programs other than the specializer.

5.2 *Encoding issues*

Unlike offline specializers for untyped languages, which need not encode values, 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. This reduces the speed of the program generator, and of the program generator generation process; for example, specializing TINY on the MP interpreter with FUSE required 7 minutes and a 32 megabyte heap.

There are several ways in which we might reduce this overhead. The generated program generator often encodes values unnecessarily, encoding values whose binding times are never examined. Current *arity raising* (Romanenko, 1990) 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 (Steele Jr., 1978) 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 (Henglein, 1992; Peterson, 1989) could provide significant improvements here, not only for the specialization of specializers, but the specialization of interpreters for dynamically typed languages as well. Launchbury’s lazy encoding technique for typed languages (Launchbury, 1991) 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.

5.3 *Accurate BTA*

Another potential research direction involves the use of binding-time analysis techniques which do not force the specializer into overly general behavior. That is, the result of an operation could be static or dynamic, the result should be annotated as ‘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, yielding an an (optimized) *online* specializer. Consel describes such a binding-time lattice in (Consel, 1989) but expresses concerns over the pollution of entire expressions due to a single subexpression having an unknown binding time. Bondorf's Treemix (Bondorf, 1990) uses such an analysis, but its effectiveness is not described.

Such mechanisms may well be useful in eliminating some encoding overhead, and in preserving binding times without the more complex mechanisms of Sections 3.3 and 4. Things are complicated, however, by the need to approximate the specializer's termination mechanism at BTA time; without means to handle the problems of Section 4.3, almost all values will be annotated with unknown binding times. Also, since online specializers spend a higher proportion of their effort on operations which cannot be optimized given knowledge of binding times, there is some question as to the benefits to be gained.

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) but do not make use of binding-time information, are both larger and slower than their efficient counterparts. However, preliminary experiments conducted by the first author 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.

6 Summary

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 generation 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 decisions, 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, FUSE. Unlike binding-time approximations, which address only the specialization of specializers, the information preservation techniques used here can 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 (Ershov *et al.*, 1988; Sahlin, 1991; Lakhotia and Sterling, 1991).

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 (c.f. Section 4). 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. 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.

References

Aiken, A., & Murphy, B. R. (1991) Static Type Inference in a Dynamically Typed Language. *Pages 279–290 of: Eighteenth Annual ACM Symposium on Principles of Programming Languages.*

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

Bondorf, A. (1990) *Self-Applicable Partial Evaluation.* Ph.D. thesis, DIKU, University of Copenhagen, Denmark. Revised version: DIKU Report 90/17.

Bondorf, A., Jones, N., Mogensen, T., & Sestoft, P. (1988) *Binding Time Analysis and the Taming of Self-Application.* Draft, 18 pages. DIKU, University of Copenhagen, Denmark.

Consel, C. (1989) *Analyse de programmes, Evaluation partielle et Génération de compilateurs.* Ph.D. thesis, Université de Paris 6, Paris, France. 109 pages. (In French).

Consel, C., & Danvy, O. (1990) From Interpreting to Compiling Binding Times. *Pages 88–105 of:* Jones, N. (ed), *Proceedings of the 3rd European Symposium on Programming.* Springer-Verlag, LNCS 432.

Ershov, A. P. (1977) On the Partial Computation Principle. *Information Processing Letters*, **6**(2), 38–41.

Ershov, A. (1978) On the Essence of Compilation. *Pages 391–420 of:* Neuhold, E. (ed), *Formal Description of Programming Concepts.* North-Holland.

Ershov, A., Bjørner, D., Futamura, Y., Furukawa, K., Haraldson, A., & Scherlis, W. (eds). (1988) *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.

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

Glück, R. (1991) Towards Multiple Self-Application. *Pages 309–320 of: Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (SIGPLAN Notices, vol. 26, no. 9, September 1991).* ACM.

Glück, R., & Turchin, V. F. (1989) *Experiments with a self-applicable supercompiler.* Tech. rept. City University of New York.

Gomard, C., & Jones, N. (1991) A Partial Evaluator for the Untyped Lambda-Calculus. *Journal of Functional Programming*, **1**(1), 21–69.

Haraldsson, A. (1977) *A Program Manipulation System Based on Partial Evaluation.* Ph.D. thesis, Linköping University. Published as Linköping Studies in Science and Technology Dissertation No. 14.

Henglein, F. (1992) Global Tagging Optimization by Type Inference. *Pages 205–215 of: Proceedings of the 1992 ACM Conference on Lisp and Functional Programming (LISP Pointers vol. 5, no. 1, January-March 1992).*

Holst, C. K. (1991) Finiteness Analysis. *Pages 473–495 of: Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (LNCS 523)*Springer-Verlag, for ACM.

Holst, C. K., & Launchbury, J. (1991) Handwriting Cogen to Avoid Problems with Static Typing. *Pages 210–218 of: Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming, Skye, Scotland.* Glasgow University.

Jones, N. D., Sestoft, P., & Søndergaard, H. (1985) An experiment in partial evaluation: The generation of a compiler generator. *Pages 124–140 of: Rewriting Techniques and Applications.* Springer-Verlag, LNCS 202.

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

Jones, N. D., Gomard, C., & Sestoft, P. (1993) *Partial Evaluation and Automatic Program Generation.* (in progress).

Kahn, K. M. (1982) A partial evaluator of Lisp programs written in Prolog. *Pages 19–25 of:* Caneghem, M. V. (ed), *First International Logic Programming Conference.*

Katz, M., & Weise, D. (1992) Towards a New Perspective on Partial Evaluation. *Pages 29–37 of: ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Directed Program Manipulation.* Proceedings available as YALEU/DCS/RR-909.

Lakhotia, A., & Sterling, L. (1991) ProMiX: A Prolog Partial Evaluation System. *Chap. 5, pages 137–179 of:* Sterling, L. (ed), *The Practice of Prolog.* MIT Press.

Launchbury, J. (1991) Self-Applicable Partial Evaluation without S-Expressions. *Pages 145–164 of: Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (LNCS 523)*Springer-Verlag, for ACM.

Mogensen, T. (1989) *Binding Time Aspects of Partial Evaluation.* Ph.D. thesis, DIKU, University of Copenhagen, Copenhagen, Denmark.

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

Rees, J., Clinger, W., *et al.*. (1991) Revised^4 Report on the Algorithmic Language Scheme. *LISP Pointers,* **4**(3), 1–55.

Romanenko, S. (1990) Arity Raiser and Its Use in Program Specialization. *Pages 341–360 of:* Jones, N. (ed), *ESOP '90. 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990. (Lecture Notes in Computer Science, vol. 432).* Springer-Verlag.

Ruf, E. (1993) *Topics in Online Partial Evaluation.* Ph.D. thesis, Stanford University, Stanford, CA. Published as Stanford Technical Report CSL-TR-93-563, March 1993.

Ruf, E., & Weise, D. (1992a) *Opportunities for Online Partial Evaluation.* Tech. rept. CSL-TR-92-516. Computer Systems Laboratory, Stanford University, Stanford, CA.

Ruf, E., & Weise, D. (1992b) *Preserving Information during Online Partial Evaluation.* Tech. rept. CSL-TR-92-517. Computer Systems Laboratory, Stanford University, Stanford, CA.

Sahlin, D. (1991) *An Automatic Partial Evaluator for Full Prolog.* Ph.D. thesis, Kungliga Tekniska Högskolan, Stockholm, Sweden. Report TRITA-TCS-9101, 170 pages.

Schooler, R. (1984) *Partial Evaluation as a means of Language Extensibility.* M.S. thesis, MIT, Cambridge, MA. Published as MIT/LCS/TR-324.

Steele Jr., G. L. (1978) *Rabbit: A Compiler for Scheme.* Tech. rept. AI-TR-474. MIT Artificial Intelligence Laboratory, Cambridge, MA.

Turchin, V. F. (1986) The Concept of a Supercompiler. *ACM Transactions on Programming Languages and Systems,* **8**(3), 292–325.

Turchin, V. F. (1988) The Algorithm of Generalization in the Supercompiler. *Pages 531–549 of:* Bjørner, D., Ershov, A. P., & Jones, N. D. (eds), *Partial Evaluation and Mixed Computation.* North-Holland.

Weise, D., Conybeare, R., Ruf, E., & Seligman, S. (1991) Automatic Online Partial Evaluation. *Pages 165–191 of:* Hughes, J. (ed), *Functional Programming Languages and Computer Architecture (LNCS 523).* Cambridge, MA: Springer-Verlag, for ACM.

Young, J., & O'Keefe, P. (1988) Experience with a type evaluator. *Pages 573–581 of:* Bjørner, D., Ershov, A. P., & Jones, N. D. (eds), *Partial Evaluation and Mixed Computation.* North-Holland.