

A Tour of Schism: A Partial Evaluation System For Higher-Order Applicative Languages*

Charles Consel
Pacific Software Research Center
Oregon Graduate Institute of Science & Technology
(consel@cse.ogi.edu)

1 Introduction

This paper presents a partial evaluation system, for pure, higher-order, applicative languages. This system named Schism, is based on a polyvariant binding-time analysis that treats higher-order functions as well as data structures. The key innovations of Schism can be summarized as follows.

- Schism is organized as a *back-end* partial evaluator. It processes a core language general enough to capture various applicative programming languages such as pure instances of Scheme [Ce91] and ML [MTH90].
- Annotations driving the treatment of function calls are generated automatically by default. Schism's annotation strategy addresses the polyvariance of the binding-time analysis by generating conditional annotations. That is, annotations drive the specialization phase depending on the binding-time context.
- A simple module facility allows the user to write a program in different files, and to create and use function libraries. Besides easing the programming stage, this facility makes it possible to split the preprocessing phase into two parts to factorize computations. The first part consists of the treatments local to each module (*e.g.*, simplification, automatic annotation). These treatments are performed only once and the processed module is used every time it is included in a stand-alone program. The second part includes the phases that perform global computations such as binding-time analysis.
- Schism is based on a higher-order polyvariant binding-time analysis. Unlike Similix's binding-time analysis [Bon91] and a polyvariant instance

of it [RG92, GR92], this analysis combines both a control flow and a data flow analyses. As a result, binding-time analysis is performed in one phase and only requires one fixpoint iteration. This proved to be crucial for accurate binding-time descriptions.

- A re-sugaring phase is provided that produces a high-level representation of residual programs. Therefore, the proper functionality of partial evaluation is fulfilled, that is, it is a *true source-to-source program transformation*.

Related Work

Lambda-Mix was the first binding-time based partial evaluator for a higher-order, applicative language [JGB⁺90, Gom92]. It includes on a monovariant binding-time analysis based on type inference [Gom90]. This partial evaluator handles a simple language and does not deal with partially static data.

This breakthrough was followed by two binding-time based partial evaluators: Similix [Bon91, Bon92] and Schism [Con90]. Each system is an extension of a first-order partial evaluator [BD91, Con89]. They both treat a subset of Scheme, and are based on a monovariant binding-time analysis that deals with data structures. Similix handles restricted side-effects on variables.

Recently, Gengler and Rytz have extended Similix's binding-time analysis to a polyvariant one [GR92, RG92]. This extension comes at the expense of duplicating code and iterating Similix's control flow and binding-time analyses.

Unlike Gengler and Rytz's extension of Similix, the new version of Schism is based on a binding-time analysis that combines both a control flow and a data flow analyses. This makes it possible to have a single fixpoint iteration [Con93]. Also, Schism is now organized so as to accept various applicative languages. A high-level representation of residual programs is produced by re-sugaring them into the original source language. Schism includes a phase that automatically generates annotations for functions, independently of their binding-time properties.

*This research is supported by NSF grant CCR-9224375.

Plan

This paper is structured as follows. In Section 2, the programming language handled by Schism is examined. In Section 3, we explain in what sense Schism can be seen as a back-end partial evaluator. Then, the various phases of the system are described. Following the usual structure of binding-time based partial evaluators, these phases are grouped into three categories: preprocessing (Section 4), processing/specialization (Section 5), and postprocessing (Section 6). Section 7 gives an overview of Schism’s programming environment. Finally, Section 8 gives some concluding remarks.

2 The Language

$k \in \text{Constant}, x \in \text{Identifier},$
 $e \in \text{Expression}, \text{def} \in \text{Definition},$
 $\text{an} \in \text{Annotation}$

$\text{def} ::= (\text{define } (i^*) \text{ an } e)$
 $| (\text{definePrimitive } i)$
 $e ::= k \mid x \mid (\text{if } e \ e \ e)$
 $| (\text{lambda } (i^*) \text{ an } e) \mid (e^+)$
 $\text{an} ::= (\text{filter } e \ e)$

Figure 1: The Core Language of Schism

Schism processes a set of recursive function definitions and primitive operators. The core language, shown in Figure 1, is a pure subset of Scheme. Its minimality makes the partial evaluation system simple. As discussed in Section 4.1, high-level constructs can be introduced via a syntactic extension mechanism [Con88]. The only unusual feature of this language is the construct `filter`.

Filters

In Schism, user-defined functions (and abstractions) contain a filter to guide the partial evaluation of an application [Con89]. Specifically, a filter consists of two parts: one that specifies how to transform a call to a function, and the other that controls the propagation of static arguments, and thus, how a function is specialized¹. Filter is a flexible construct: it can be used in an on-line partial evaluator [Con88] as well in an off-line one. Also, both the transformations yielded

¹For simplicity, in this paper we assume that the expressions in a filter are written in the core language of Schism (see Figure 1). In fact, in the current implementation of Schism, these expressions are written in Scheme.

by the first part of a filter and the propagation directives are definable; there could be extended. The filters considered in this paper are dedicated to off-line partial evaluation.

Because they control how static data are propagated, in an off-line partial evaluator, filters must be invoked prior to specialization during binding-time analysis. The following function illustrates the use of a filter.

```
(define (f p1 ... pn) (filter e1 e2) e3)
```

During binding-time analysis, if Function `f` is called, the first part of the filter (`e1`) is invoked with the context of the call, *i.e.*, each parameter of `f` is bound to the binding-time value of the corresponding argument. This part determines how to transform a call to `f`. It can be seen as the following function.

$$(\lambda (p_1, \dots, p_n). e_1) : B^n \rightarrow \text{Transf}$$

The filter is invoked during binding-time analysis and receives the binding-time value of each argument and not a concrete value. Predicates operating on the binding-time domain are provided so that the user can write filters. The co-domain of the first part of the filter defines the transformations to be performed on function calls. The domain *Transf* contains two values `U` and `R`, denoting unfolding and residualization, respectively.

If a call is made residual, the called function is specialized. This specialization is specified by the second part of the filter. It can be seen as the following function.

$$(\lambda (p_1, \dots, p_n). e_2) : B^n \rightarrow T^n$$

Like the first part of the filter, the second one is invoked with the context of the call. The result is a list of propagation values, each of which is `C` if the corresponding argument is static and should be propagated at compile-time, and is `R` otherwise. Examples of filters are displayed in Figure 4.

Types

To extend the expressiveness of the language we have introduced constructs to declare sum and product types, and to manipulate values of such types. Unnamed product values are constructed using `tuple`; this data constructor can also be used as the left-hand side of a let-binding clause. These constructs, displayed in Figure 2, are purely syntactic: Schism does not include a static type checker. Their similarity with ML makes it easy to translate programs written in a subset of ML to Schism’s core language, as well as translate residual programs written in Schism back to ML. This is further discussed in Section 3.

```

t ∈ TypeDeclaration
c ∈ DataConstructor
x ∈ Identifier
e ∈ Expression

t ::= (defineType c x+)
    | (defineType x (c x*))

(caseType e [(c x*) e]+)

(let [(c x*) e]+) e

```

Figure 2: Type Extensions

3 Back-End Partial Evaluation

Schism has been designed as a back-end partial evaluator. Its core language captures a class of applicative languages. Therefore, programs written in various languages can be translated into the core language for the purpose of partial evaluation. Then residual programs can be converted back into the original source language. As in traditional compilation, partial evaluation techniques are very similar within a given class of programming languages.

Currently Schism handles subsets of Scheme and ML. Scheme constructs are defined using macro-functions. ML programs are translated using CRML, a compile-time reflection system for ML [She92]. It is a subset of ML because the module system, the imperative features, and records are not handled.

Thanks to the type constructs available in Schism’s core language this subset of ML is essentially mapped directly to Schism. An example of translation is presented in Figure 3. Figure 3-(a) is the original ML program, Figure 3-(b) is its Schism translation. Translation back to Schism’s typed representation (and thus to ML) is discussed in Section 6.

4 Preprocessing

As a binding-time based partial evaluator, Schism consists of several preprocessing phases. This structure aims at lifting as many computations as possible from the specializer. Indeed, these preprocessing phases are performed only once for a given binding-time description of the input of a program. This section gives an overview of these various phases.

4.1 Expansion and Linking

As mentioned previously Schism’s language is extensible: one can introduce high-level constructs via a

syntactic extension mechanism. Similar to a macro-expansion phase in a Lisp system, Schism includes a phase aimed at expanding syntactic extensions into the core language.

To keep the system simple, the data structures manipulated by a program are represented as lists in the core language. This conversion is part of the expansion phase. When converting product and sum values we need to keep track of type information for subsequent phases such as binding-time analysis or re-sugaring of residual programs. To do so, list operators are annotated with type information.

As is customarily done in Lisp compilers (*e.g.*, [Ste78, KKR⁺86]), after macro-expansion, some simple program transformations eliminate trivial inefficiencies in the expanded program. These transformations are important because they simplify programs and thus reduce the processing of the subsequent phases noticeably. For example, we have observed that they can slash by two the number of iterations performed by the binding-time analysis.

Schism offers a simple module facility. It allows the user to organize a Schism program into separate files, and to create and use libraries (*e.g.*, for operations on lists and boolean values). Expansion and simplification of programs are performed on each module. These phases correspond to the first stage of Schism’s preprocessing component. The second stage consists of creating a stand-alone program for the purpose of specialization. This is done by invoking the linker with the necessary modules and the main function. This phase produces a program that only includes the necessary functions and primitives. This new division of the preprocessing component makes it possible to factorize computations and improve efficiency.

4.2 Automatic Filter Generation

The system includes a phase that automatically annotates functions with filters. The flexibility of this construct makes it possible to use various annotation strategies. In the current version of Schism, we have chosen the automatic annotation strategy of Similix [BD91]. This strategy lies on the following observation. Any cycle in a control flow graph of a program contains at least one conditional to determine whether or not to recurse². If this test expression is static, it is safe to unfold a recursive call. However, if the test expression is dynamic, unfolding a recursive call may cause non-termination. Thus, the idea is to create a new function for this dynamic conditional expression so as to prevent unfolding infinitely.

More specifically, a combinator is extracted out of each dynamic conditional expression. This conditional expression is replaced by a call to the new combinator.

²This observation assumes that programs are written in a call-by-value language.

<pre> datatype tree = Leaf of int Branch of tree * tree; fun sum (t1, t2) = case t1 of Leaf n1 => (case t2 of Leaf n2 => Leaf (n1 + n2) Branch (_,_) => Leaf 0) Branch (lt1, rt1) => (case t2 of Leaf _ => Leaf 0 Branch (lt2, rt2) => Branch (sum (lt1, lt2), sum (rt1, rt2))); fun main (t1, t2) = sum (t1, t2); </pre>	<pre> (defineType tree (Leaf x1) (Branch x1 x2)) (define (sum t1 t2) (caseType t1 [(Leaf n1) (caseType t2 [(Leaf n2) (Leaf (+ n1 n2))] [(Branch - -) (Leaf 0)])]) [(Branch lt1 rt1) (caseType t2 [(Leaf -) (Leaf 0)] [(Branch lt2 rt2) (Branch (sum lt1 lt2) (sum rt1 rt2))])])]) (define (main t1 t2) (sum t1 t2)) </pre>
(a) Program Written in ML	(b) Translated Version in Schism

Figure 3: Program that Sums Two Trees

<pre> (define (member v l) (if (null? l) '#f (if (equal? v (car l)) '#t (member v (cdr l)))))) </pre>	<pre> (define (member v l) (filter (if (stat? l) U R) (list C C)) (if (null? l) '#f (member-1 v l))) (define (member-1 v l) (filter (if (every? stat? (list v l)) U R) (list C C)) (if (equal? v (car l)) '#t (member v (cdr l)))) </pre>
(a) Unannotated	(b) Annotated

Figure 4: Automatic Generation of Filters for Function member

This combinator becomes a new *specialization point* in that the specializer is instructed to specialize any call to it.

Similix's strategy is attractive because it does not require any control flow information unlike others (*e.g.*, [Ses88, Hol91]). Also, it handles first-order as well as higher-order programs. Its main drawback is that does not address infinite specialization [BD91].

In the context of filters, this strategy needs to be somewhat adapted. Indeed, filters are given prior to binding-time analysis. Also, a filter is a conditional annotation in that it may yield a different transformation and different propagation directives depending on the binding-time value of the function-call arguments. In fact, this feature is crucial in the context of a polyvariant binding-time analysis: each a function is called with dif-

ferent binding-time values and it should be transformed differently depending of these values.

The above observations are reflected in the Schism's version of Similix's automatic annotation strategy. It can be outlined as follows. For a given conditional expression, we first collect the free variables that occur in the test expression. The binding-time value of these variables will determine whether or not unfolding may cause non-termination. Then, we extract a combinator out of this conditional expression. This new function is attached a filter that specifies that any call to this function must be unfolded only if all the free variables in the test expression are static. Otherwise this function must be specialized. Notice that inner conditional expressions may not yield new combinators if their test expression consists of the same free variables as the ones in the outermost conditional expression.

Figure 4 illustrate Schism’s annotation strategy with the example of Function `member`³. Notice that the first conditional expression does not yield a specialization point because `member` is a function, and thus, it is already a specialization point. The inner conditional expression produces a new specialization point because its test expression contains an additional variable `v`.

The automatic filter generation phase has been successfully used on such programs as interpreters and string and pattern matching programs. This phase introduces very little overhead because it only requires one traversal of a program to generate the filters and the additional specialization points. Furthermore, since our annotation strategy is local to a function, and thus independent of its binding-time properties or sites where it is applied, filters are generated only once for a given module (after expansion and simplification).

4.3 Binding-Time Analysis

The binding-time analysis takes a stand-alone program and a binding-time description of the input and produces binding-time signatures (*i.e.*, binding-time descriptions) for the functions, the abstractions, and the data structures in the program. Schism’s binding-time analysis is polyvariant and makes use of type information, when available, to produce more accurate information. A detailed description of this analysis is available elsewhere [Con93].

The binding-time signatures of a program are used to create binding-time trees for each function. A binding-time tree is isomorphic to the abstract syntax tree of a function and contains a binding-time value for each expression in a function. Binding-time trees are introduced to facilitate the subsequent treatments. Notice that since the analysis is polyvariant a given function may have more than one binding-time tree.

4.4 Action Analysis

Based on the binding-time trees of a program, this phase determines a *specialization action* (*i.e.*, program transformation) for each expression in the program. This analysis aims at lifting the interpretation of the binding-time values of a program from the specialization phase. This is particularly important when the binding-time analysis uses rich abstract domains [Con89, CD90]. This strategy simplifies drastically the specialization phase and improves its efficiency.

A set of actions is defined for each construct. It corresponds to the different binding-time contexts that can occur for a given construct. The simplest actions that are actually defined for every construct are `Id` and `Ev`. The first one is assigned to an expression that does not manipulate any available data at partial-evaluation

³Function `member` tests whether a value is an element of a list.

time. In this case the expression is reproduced verbatim. The second action is assigned to an expression that only manipulates available data. The corresponding treatment consists of fully evaluating the expression. Both actions improve the specialization process because they can be assigned to large expressions in a program. This avoids symbolic evaluation where no treatment or standard evaluation can be performed instead.

As an example of other actions, let us consider the treatment of an application. Besides the `Id` and `Ev` actions, there are three actions for this construct: `AppRecons`, `AppI` and `AppFreeze`. The first action is assigned to an application composed of an operator that partially evaluates to a residual expression, not to a functional value. Action `AppI` corresponds to an application whose operator partially evaluates to a functional value. Finally, `AppFreeze` is similar `AppI` but the application itself partially evaluates to a functional that needs to be frozen because it is surrounded by a residual expression, and thus not applied until run time. Note that we could refine the treatment of applications. For example, we could introduce a new action that is assigned to first-order application. As a result, such an application would require a simpler treatment of specialization: the operator would not need to be partially evaluated. This new action would eliminate the creation of closures to treat first-order applications.

In fact, specialization actions can be ordered with respect to the binding-time context they require. In developing the set of actions, one may initially define few actions; the missing ones can be mapped to some more conservative actions.

The action analysis produces a program annotated with actions whose type is displayed in Figure 5-(a). The annotated syntax also contains the information necessary to perform a given action. Notice that `Ev` and `Id`-expressions are represented with an unannotated syntax as show in Figure 5-(b). This is because they require either no treatment or standard evaluation.

5 Specialization

The specializer is passed an annotated program and specialization values. Since each expression is annotated with an action, the main task of the specializer is to dispatch on the action as shown in Figure 6. Function `spec` has type

$$\begin{aligned} \text{Spec} &: AExpr \times Env \times Prog \times Cache \rightarrow \text{SpecAnswer} \\ \text{where } \text{SpecAnswer} &= \text{SpecValue} \times Cache \\ \text{SpecValue} &= (Expr + (\text{SpecValue}^n \rightarrow \text{SpecValue})) \\ &\quad \times \text{SymbVal} \end{aligned}$$

For readability, we only display direct style excerpts of the specializer. However, in practice a specializer written in continuation-passing style (CPS) performs better because it exposes more static computations and achieves part of the effect of converting a program into CPS [CD91]. Similix has also adopted this strategy [Bon92].

```

(defineType AExpr
  (Id Expr)
  (Ev Expr)
  (AType Symbol)
  (ParLookup Symbol)
  (ParLookupFreeze Symbol)
  (IfReduce AExpr AExpr AExpr)
  (IfRecons AExpr AExpr AExpr)
  (IfFreeze AExpr AExpr AExpr)
  (OpSelect IdValue)
  (AbsSelect AbsId StatFreeVars LiftFreeVars)
  (AbsFreeze AbsId)
  (Appl InstPat PropPat SbtPat AExpr AExprs)
  (AppFreeze InstPat PropPat SbtPat AExpr AExprs)
  (AppRecons AExpr AExprs))

(defineType Expr
  (Constant Value)
  (Type Symbol)
  (Identifier IdValue)
  (Conditional Expr Expr Expr)
  (Application Expr Exprs)
  (Abstraction Parameters Expr))

(a) Annotated Syntax
(b) Standard Syntax

```

Figure 5: Abstract Syntax

```

(define (spec ae env prog cache)
  (caseType ae
    [(Id e) (specId e cache)]
    [(Ev e) (specEv e env prog cache)]
    [(AType tn) (specType tn cache)]
    [(ParLookup n) (dParLookup n env cache)]
    [(ParLookupFreeze n) (dParLookupFreeze n env cache)]
    [(IfReduce ae1 ae2 ae3) (dIfReduce ae1 ae2 ae3 env prog cache)]
    [(IfRecons ae1 ae2 ae3) (dIfRecons ae1 ae2 ae3 env prog cache)]
    [(IfFreeze ae1 ae2 ae3) (dIfFreeze ae1 ae2 ae3 env prog cache)]
    [(OpSelect idValue) (dOpSelect idValue prog cache)]
    [(AbsSelect absId statFreeVars liftFreeVars)
     (dAbsSelect absId statFreeVars liftFreeVars env prog cache)]
    [(AbsFreeze absId) (dAbsFreeze absId env prog cache)]
    [(Appl instPat propPat sbtPat ae aes)
     (dAppl instPat propPat sbtPat ae aes env prog cache)]
    [(AppFreeze instPat propPat sbtPat ae aes)
     (dAppFreeze instPat propPat sbtPat ae aes env prog cache)]
    [(AppRecons ae aes) (dAppRecons ae aes env prog cache)]))

```

Figure 6: Main Specialization Function

Function `spec` is passed an annotated expression, an environment, the complete program, and a cache that contains the specialized functions. It returns a specialization value and a cache. The first component of a specialization value is either a residual expression or a function. The latter occurs when an expression partially evaluates to a higher-order value. The second component of a specialization value is a symbolic value. It is used when an expression partially evaluates to partially static data or a higher-order value. For the former it contains the static parts of the data; this facilitates simplifications of list expressions. For a higher-order value, the symbolic value consists of the function label (indeed, each function or abstraction is uniquely identified) and the static and dynamic free variables of the function. This information is used when specializing a function with respect to higher-order values.

Figure 7 displays the specialization treatments of an application. As can be seen, these treatments are simple. This is because all decisions have been taken prior to specialization by the action analysis and thus the specialization process has been decomposed in fine-grained specialization actions. Notice that, in contrast with Similix [Bon91] the specializer is written with higher-order functions, whether or not they are evaluated at partial-evaluation time. Therefore, an operator partially evaluates to a functional value which is applied to the partially evaluated arguments.

Because Schism is based on a polyvariant binding-time analysis, an operator may occur in different contexts, namely, it can be applied or frozen. When it is applied the binding-time value of the arguments may vary depending on the application site. To handle these cases the functional value yielded by an operator is abstracted over both its usage (*i.e.*, `Applied` and `Frozen`), and the binding-time value of the arguments `instPat`. This last component is used to select the appropriate annotated function body.

6 Postprocessing

This part of Schism consists of two phases: a simplification phase and a re-sugaring phase. The simplification phase performs some obvious optimizations that are not worth including in the specializer because they do not trigger further static computations, and are done only once.

The second phase of the postprocessing part aims at re-sugaring the residual programs using standard syntactic extensions (*e.g.*, `and`, `or`) and user-defined data constructors. This phase is important because it allows the user to examine a high-level representation of residual programs. This makes it possible to check whether the degree of specialization of a program corresponds to what was expected.

To illustrate this phase let us come back to the `Sum` program displayed in Figure 3 and let us specialize it with respect to the tree (`Branch (Leaf 1) (Leaf 2)`).

Figure 8-(a) presents the re-sugared version of the residual program in Schism. This re-sugared version corresponds exactly to what is expected: all the type operations have been reconstructed, including necessary variables for the pattern matching in the `caseType`. As a result, the residual program can easily be examined to check whether all the computations expected to be static have been performed. Figure 8-(b) shows the translation back in ML.

It could be argued that when being expanded, syntactic extensions should be attached to the resulting expressions to enable a re-sugaring more faithful to the original program. However, this would complicate significantly the subsequent phases and it still does not seem to ensure that original syntactic extensions could always be retrieved. Based on our experience, the re-sugaring phase we have developed produces a very readable and high level representation of residual programs.

The main goal of the re-sugaring phase, as well as the other programming tools discussed in the next section, is to help the user pinpoint unexpectedly dynamic computations. Once a problem is found the user needs to modify the program to improve its binding-time behavior. Experience shows that the main class of problems can be solved by converting the original program (or parts of it) in continuation-passing style [CD91]. Note that this transformation does not compromise the readability of the residual program since it can be converted back to direct style [Dan92].

7 Programming Environment

The programming environment that is currently implemented in Schism addresses three stages of partial evaluation: the binding-time analysis, the specialization phase, and the residual program. The last stage is covered by the previous section and consists of providing the user with a high-level representation of residual programs via re-sugaring.

Regarding the binding-time analysis, Schism's programming environment provides an *inspector* of binding-time values. This tool consists of operations to display binding-time signatures for functions, abstractions and data structures. In contrast with other similar tools [Pai90, Mos91], this inspector provides support to deal with polyvariant descriptions of objects. Furthermore, to help the user coping with rich and extensive binding-time information, the inspector offers various levels of details for each display operation. For example, one level consists of printing binding-time signatures using a type language and user-defined types (*i.e.*, product and sum types), when possible.

Besides inspecting binding-time information, one can also monitor the binding-time analysis. This is important since this analysis is based on a fixpoint iteration and thus the resulting binding-time information does not give the context in which a variable becomes dynamic. Monitoring the binding-time analysis process

```

(define (dAppI instPat propPat sbtPat ae aes env prog cache)
  (let ([[SpecAnswer fv - cache) (spec ae env prog cache)]])
    (specArgs aes env prog cache
      (lambda (spvs cache)
        ((fv (Applied)) instPat propPat sbtPat spvs cache))))))

(define (dAppRecons ae aes env prog cache)
  (let ([[SpecAnswer eOp - cache) (spec ae env prog cache)]])
    (specArgs aes env prog cache
      (lambda (spvs cache)
        (SpecAnswer
          (Application eOp (mapcar
            (lambda (spv) (let ([[SpecValue e -] spv]) e))
              spvs))
          (SVUnit) cache))))))

(define (dAppFreeze instPat propPat sbtPat ae aes env prog cache)
  (let ([[SpecAnswer fv - cache) (dAppI instPat propPat sbtPat ae aes env prog cache)]])
    ((fv (Frozen)) cache)))

```

Figure 7: Treatments of an Application

<pre> (define (main.1 t2) (caseType t2 [(Leaf -) (Leaf '0)] [(Branch x1 x2) (Branch (caseType x1 [(Leaf x1.19) (Leaf (+ '1 x1.19))] [(Branch - -) (Leaf '0)]) (caseType x2 [(Leaf x1) (Leaf (+ '2 x1))] [(Branch - -) (Leaf '0)])))])) </pre> <p>(a) Schism Residual Program</p>	<pre> fun main_1 t2 = case t2 of Leaf _ => Leaf 0 Branch (x1, x2) => Branch ((case x1 of Leaf x1_19 => Leaf (1 + x1_19) Branch (_, _) => Leaf 0), (case x2 of Leaf x1 => Leaf (2 + x1) Branch (_, _) => Leaf 0)); </pre> <p>(b) Translated Version in ML</p>
---	--

Figure 8: Sum Program Specialized w.r.t. (Branch (Leaf 1) (Leaf 2))

is achieved by introducing *binding-time declarations*. That is, just like type declarations, one can declare the binding time properties of functions, abstractions, and data structures. These declarations are checked during the binding-time analysis. If they are violated (that is, in the lattice of abstract values, a value is greater than the one declared), the binding-time analyzer presents the user with contextual information (that is, the expression that caused the violation).

Regarding the specialization phase, Schism's programming environment is concerned with non-termination. It provides a tracing mechanism that allows one to monitor the unfolding and the specialization of function calls.

This programming environment and future extensions are described in [CP92].

8 Conclusion

This paper has presented Schism, a partial evaluation system for higher-order, applicative languages. This system is based on a polyvariant binding-time analysis that deals with higher-order functions as well as data structures. Schism is a back-end partial evaluator in that it processes a core language rich enough to capture various applicative programming languages both typed and untyped. Schism includes a re-sugaring phase that makes it possible to perform true source-to-source program transformation.

Schism is structured as a self-applicable partial evaluator. Based on our previous experience in developing self-applicable partial evaluators for first-order as well as higher-order programs [Con89, Con90], this structure should make it possible to self-apply this new system. Because specialization is already optimized via the introduction of actions, we do not expect a speed-up comparable to partial evaluators that do not include such a phase. Indeed, the action phase performs part of the computations usually achieved by self-application.

References

- [BD91] A. Bondorf and O. Danvy. Automatic auto-projection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.
- [Bon91] A. Bondorf. Automatic autoprojection of higher-order recursive equations. *Science of Computer Programming*, 17:3–34, 1991.
- [Bon92] A. Bondorf. Improving binding times without explicit CPS-conversion. In *ACM Conference on Lisp and Functional Programming*, pages 1–10, 1992.
- [CD90] C. Consel and O. Danvy. From interpreting to compiling binding times. In N. D.

Jones, editor, *ESOP'90, 3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 88–105. Springer-Verlag, 1990.

- [CD91] C. Consel and O. Danvy. For a better support of static data flow. In *FPCA'91, 5th International Conference on Functional Programming Languages and Computer Architecture*, pages 496–519, 1991.
- [Ce91] W. Clinger and J. Rees (editors). Revised⁴ report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July–September 1991.
- [Con88] C. Consel. New insights into partial evaluation: the Schism experiment. In H. Ganzinger, editor, *ESOP'88, 2nd European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 236–246. Springer-Verlag, 1988.
- [Con89] C. Consel. *Analyse de Programmes, Evaluation Partielle et Génération de Compilateurs*. PhD thesis, Université de Paris VI, Paris, France, June 1989.
- [Con90] C. Consel. Binding time analysis for higher order untyped functional languages. In *ACM Conference on Lisp and Functional Programming*, pages 264–272, 1990.
- [Con93] C. Consel. Polyvariant binding-time analysis for higher-order, applicative languages. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 1993. To appear.
- [CP92] C. Consel and S. Pai. A programming environment for binding-time based partial evaluators. In C. Consel, editor, *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 62–66. Yale University, 1992. Research Report 909.
- [Dan92] O. Danvy. Back to direct style. In B. Krieg-Brückner, editor, *ESOP'92, 4th European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*, pages 130–150. Springer-Verlag, 1992.
- [Gom90] C. K. Gomard. Partial type inference for untyped functional programs. In *ACM Conference on Lisp and Functional Programming*, pages 282–287, 1990.
- [Gom92] C. K. Gomard. A self-applicable partial evaluator for the lambda-calculus: Correctness and pragmatics. *ACM Transactions on Programming Languages and Systems*, 14(2):147–172, 1992.

- [GR92] M. Gengler and B. Rytz. A polyvariant binding time analysis handling partially known values. In *Workshop on Static Analysis*, volume 81-82 of *Bigre Journal*, pages 322–330. IRISA, Rennes, France, 1992.
- [Hol91] C. K. Holst. Finiteness analysis. In *FPCA'91, 5th International Conference on Functional Programming Languages and Computer Architecture*, pages 473–495, 1991.
- [JGB⁺90] N. D. Jones, C. K. Gomard, A. Bondorf, O. Danvy, and T. Mogensen. A self-applicable partial evaluator for the lambda calculus. In *IEEE International Conference on Computer Languages*, pages 49–58, 1990.
- [KKR⁺86] D. A. Kranz, R. Kelsey, J. A. Rees, P. Hudak, J. Philbin, and N. I. Adams. Orbit: an optimizing compiler for Scheme. *SIGPLAN Notices, ACM Symposium on Compiler Construction*, 21(7):219–233, 1986.
- [Mos91] C. Mossin. Similix binding time debugger manual. Technical report, University of Copenhagen, Copenhagen, Denmark, 1991.
- [MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of ML*. MIT Press, 1990.
- [Pai90] S. Pai. Some user interface enhancements to Schism. Technical report, Yale University, New Haven, Connecticut, USA, 1990. (Student Report).
- [RG92] B. Rytz and M. Gengler. A polyvariant binding time analysis. In C. Consel, editor, *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 21–28. Yale University, 1992. Research Report 909.
- [Ses88] 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*. North-Holland, 1988.
- [She92] T. Sheard. A guide to CRML: a compile-time reflective ML. Research report, Pacific Software Research Center, Oregon Graduate Institute of Science and Technology, Beaverton, Oregon, USA, 1992.
- [Ste78] G. L. Steele. Rabbit: A compiler for scheme. Master's thesis, M.I.T (A.I. LAB.), Massachusetts, U.S.A, 1978.