From Interpreting to Compiler Binding Times

Charles Consel and Olivier Danvy
Research Report YALEU/DCS/RR-774
March 1990

This work is supported by the Darpa grant N00014-88-K-0573.

From Interpreting to Compiling Binding Times *

Charles Consel † & Olivier Danvy ‡

Abstract

The key to realistic self-applicable partial evaluation is to analyze binding times in the source program, *i.e.*, whether the result of partially evaluating a source expression is static or dynamic, given a static/dynamic division of the input. Source programs are specialized with respect to the static part of their input. When a source expression depends on the concrete result of specializing another expression, the binding time of this other expression is first interpreted. A safe approximation of these abstract values is computed by binding time analysis.

This paper points out that this value-based information can be compiled into control-based directives driving the specializer as to what to do for each expression – instead of how to use the result of partially evaluating an expression. This compilation is achieved by a non-standard interpretation of the specialization semantics, based on the observation that a source expression is either reduced or rebuilt. The result is an action trees isomorphic to the abstract syntax tree of the source program. This approach suggests to reorganize the specializer so that it is driven first by the action tree and then by the abstract syntax tree – instead of performing first a syntax analysis and then interpreting binding times.

Some subtrees imply the corresponding expressions to be completely reduced or completely rebuilt. These expressions are completely evaluated or reproduced verbatim. This suggests to refine the specializer so that it evaluates, reduces, rebuilds, or reproduces source expressions. This also suggests a more radical implementation.

By pruning the source program based on its action trees, we extract combinators for each subtree that should be evaluated completely or reproduced verbatim. By implementing these combinators as runtime operators, we prune the text of the partial evaluator that was dedicated to evaluating or reproducing source expressions using symbolic interpretation. Because source programs are smaller, they can be specialized faster. Because half of the specializer disappears, it is smaller and faster too. For these compound reasons self-application performs better.

As a result, more processing is shifted away from the actual specializer. A pleasing symmetry appears in the actual specialization process. Its significance, *e.g.*, in the structure of compilers generated by self-application, remains to be explored.

Keywords: partial evaluation, self-application, actions, combinators.

*Extended abstract to appear in the proceedings of ESOP 1990.

†Yale University (consel-charles@cs.yale.edu). This work was done jointly at University of Paris 6 under a MRES grant and at Yale under the Darpa grant N00014-88-K-0573.

‡This work has been carried out during a visit to the Computer Science Department of Indiana University, in the fall of 1989 (danvy@iuvax.cs.indiana.edu).

Introduction

Partial evaluation is a program transformation specializing a program with respect to a static part of its input. For self-application purposes, binding times in the source program are analyzed prior to the actual specialization. A binding time analysis automatically computes the binding time values (static or dynamic) of each expression in the source program with respect to an abstraction of its input: the arguments that are available are declared static and the others are dynamic. The result of binding time analysis is a collection of binding time trees isomorphic to the source abstract syntax trees (one for each procedure). Then the source program and the binding time trees are processed by the specializer together with concrete values. Specializing a source expression is achieved by first determining its syntactic category and then interpreting the binding time value of its sub-components as the result of their specialization is needed.

The point of this paper is that this interpretation of binding time values can be lifted from the specializer. Let us illustrate this through an example. Figure 1 displays a (first-order) program written in Scheme [Rees & Clinger 86] that computes the function concatenating two lists.

Our goal is to specialize procedure **append** with respect to its second argument. The result of preprocessing (which includes parsing and binding time analysis) is displayed in figure 2.

At the top, the program is parsed. Below, the binding time tree is reproduced. It is isomorphic to the corresponding syntax tree: the node corresponding to a dynamic identifier is `(d)`, and to a static identifier is `(s)`. In general, the first component of a binding time tree is the binding time value of the expression, and the rest (if any) are the binding time trees of the corresponding sub-expressions.

Because its induction variable is dynamic in this example, procedure **append** is a specialization point. This means that all the calls to **append** will give rise to specializing **append** with respect to the value of its second argument. Also, the dynamic parameter `x` need not be renamed during specialization, while the binding of `y` will be unfolded.

Let us specialize procedure **append** with respect to its second argument whose value is `(3 4 5)`. The result is displayed in figure 3.

Let us analyze each step of the specialization. Because **append** is a specialization point, it gives rise to generating a residual procedure, named

```
(define append   ;;; List(A) * List(A) -> List(A)
  (lambda (x y)
    (if (null? x)
        y
        (cons (car x)
              (append (cdr x)
                      y)))))
```

Figure 1: Tagged concatenation of lists (source program)

```
(
 (append
  (*lambda-sp (x y) [i u]
    (*if (*app null? (*ide x))
         (*ide y)
         (*app cons (*app car (*ide x))
                    (*app append (*app cdr (*ide x))
                                 (*ide y))))))
)
(
 (append
  (s d)
  (d (d (d))
     (s)
     (d (d (d)) (d (d (d))
                   (s)))))
)
```

Figure 2: Parsed and binding time analyzed source program
`append` is a specialization point (`lambda-sp`). Its first parameter is dynamic and left identical. Its second parameter is unfoldable. This is indicated by the tags `[i u]`.

`append-0`, which is a version of `append` specialized with respect to the static value `(3 4 5)`. In a symbolic environment where `x` is bound to the residual identifier `x` and `y` is bound to the value `(3 4 5)` represented by `'(3 4 5)`, we are ready to interpret the body of `append` symbolically.

It is a conditional expression. This requires to interpret the test expression in the same environment, which in turn needs interpreting the identifier `(*ide x)`. The result is the residual identifier `x`. Because the binding time value of the argument of `null?` is dynamic, the application is rebuilt and the result of specializing the test part is `(null? x)`. Because the binding time value of the test is dynamic, specializing the conditional expression will yield a residual conditional expression whose test part is `(null? x)`, and whose consequent and alternative parts are the result of specializing the source consequent and alternative parts.

Identifier `(*ide y)` is bound in the environment. Its interpretation yields `'(3 4 5)`.

Interpreting `(*app cons ... ...)` needs interpreting both arguments. Interpreting the first needs interpreting `(*ide x)`. Because its binding time value is dynamic, the `car` operation cannot be folded and the application is rebuilt instead. The arguments of the call to `append` are interpreted similarly. The results are two residual expressions. Because their static projection (*i.e.*, the pattern of static values in these arguments) coincides with the original one, we build a residual call to `append-0`. This call has only one argument because `append-0` is a unary procedure. At this point we return two residual expressions as actual arguments of the application `(*app cons ... ...)`. Because the binding time values of the two arguments are dynamic, the application is rebuilt.

Then the conditional expression is rebuilt, which concludes specializing procedure `append`.

To summarize: partial evaluation is staged in two phases – preprocessing and specialization. Binding time information has been collected during preprocessing. Decisions as to how to treat the program are taken during specialization, *i.e.*, which expression should be reduced, which should be rebuilt. This staging has been pioneered in [Jones *et al.* 89]. The remaining overhead (*e.g.*, interpreting the binding time informations) is removed by self-applying the specializer.

However, aside from self-application, something basic can be done: given

the program and the binding time tree, we can infer which actions the specializer will be performing. The basic action of a specializer is to reduce or rebuild a source expression. Let us build an *action tree* isomorphic to the binding time tree and tagged with the two directives *Rd* (for reduce) and *Rb* (for rebuild). The result is displayed in figure 4.

We can see this action tree as providing directives to the specializer.

Our next observation is that the specializer can do better than, *e.g.*, reproducing large pieces of source program by symbolic interpretation. This suggests to introduce a directive *Id* at the root of each action subtree expressing that the source program should be rebuilt verbatim. Symmetrically, expressions that should be reduced completely can be handled by introducing a directive *Ev*. The new action tree is displayed in figure 5.

Now we may wonder why the specializer, whose basic tasks are to reduce and rebuild source expressions, and to coordinate these actions, should have the burden of evaluating or reproducing source expressions. There are faster ways to evaluate Scheme expressions – *e.g.*, by compiling them using a regular Scheme compiler. Similarly, there are faster ways to implement the identity function.

So let us prune the action tree by extracting *Ev*- and *Id*-combinators from the source program. The **append** example is too simple for giving rise to actually defining interesting combinators, but in an interpreter, many combinators arise naturally, *e.g.*, *Ev*-combinators for scope resolution and *Id*-combinators for store management. As a result, source programs are physically smaller. They can be expected to be specialized faster, besides the fact that there is no interpretation overhead anymore.

To conclude: by abstracting the various treatments of the specializer for each syntactic construct, we have deduced four basic actions: *Ev*, *Rd*, *Rb* and *Id*. *Rd* expresses that the corresponding syntactic construct will be reduced. *Ev* is a particular case of *Rd*: it denotes an expression that may be fully evaluated because it is completely static. Symmetrically, *Rb* indicates that the corresponding syntactic construct will be rebuilt. *Id* is a particular case of *Rb*: it denotes an expression that may be reproduced textually because it is completely dynamic. In the case of both *Rd* and *Rb*, subexpressions still have to be processed.

Introducing combinators has three major effects: (1) because both *Ev*-combinators and *Id*-combinators are treated more efficiently than by sym-

```
(define append-0   ;;; List(Num) -> List(Num)
  (lambda (x)
    (if (null? x)
        '(3 4 5)
        (cons (car x)
              (append-0 (cdr x))))))
```

Figure 3: Residual program corresponding to specializing `append` with respect to `(3 4 5)`

```
(
 (append
  (rb rd)
  (rb (rb (rb))
      (rd)
      (rb (rb (rb)) (rb (rb (rb))
                        (rd)))))
)
```

Figure 4: Action tree of procedure `append`

```
(
 (append
  (id ev)
  (rb (id (id))
      (ev)
      (rb (id (id)) (rb (id (id))
                        (ev)))))
)
```

Figure 5: Action tree of procedure `append` with *Ev*- and *Id*-directives

bolic interpretation, specialization is faster; (2) because these combinators are no more a part of the actual source program, the specializer has less data to process; (3) because self-application is a particular case of specialization, it benefits from points (1) and (2).

It is interesting to notice that these combinators capture both *purely static* and *purely dynamic* semantics of the source program. The remaining actions are the essence of specialization: reducing and rebuilding expressions, and managing control.

This paper is organized as follows. Section 1 describes the conventional specialization of source programs after binding time analysis. Section 2 presents a semantics-based derivation of action trees. Section 3 describes how to specialize source programs given their action trees. Section 4 investigates how to prune action trees and define combinators for specialization. Section 5 draws some assessments. Section 6 compares this approach with related work. Finally this work is put in perspective.

1 Specializing Programs using Binding Time Trees

This section discusses the kernel of a self-applicable specializer. As generally agreed in the partial evaluation community, all realistic self-applicable specializers (*i.e.*, stand-alone and producing non-trivial compilers from interpreters) are based on some analysis of the binding times of the program [Bondorf *et al.* 88]. Interestingly enough, front-end strategies diverge widely (regarding generalizing, automatizing, polyvariance, partially static structures, computation duplication, *etc.*). Still they all converge when it comes to the actual specialization.

No decisions are taken as to how to specialize the program. However, to ensure termination and avoid code duplication, decisions as to how to treat procedure calls and let expressions are taken prior to the actual specialization. These decisions are represented with *annotations*.

Still there are computational evidences and reasons behind them to go further than the standard binding time-based specializer.

1.1 The language: first-order recursive equations

Source and residual programs are collections of recursive definitions for first-order Scheme procedures. Scheme expressions are constants, variables, con-

ditional expressions, uncurried applications of procedures and operators, and let blocks. The reduction order is call by value.

An expression and its associated binding time tree are related as follows: the expression is paired with a binding time value that specifies whether specializing the expression will yield a static value or a residual expression.

1.2 Binding time trees

The concept of binding time here is generalized from the traditional binding time of an *identifier* (compile time, link time, run time, *etc.*) to the binding time of an *expression.* The motivation is that during partial evaluation, an expression will be reduced if it depends solely on the static part of the input, or rebuilt if it depends on the rest of the input. This point is captured by the generalization of binding times from identifiers to expressions. As a consequence it is possible to know whether the result of partially evaluating any expression will yield a static value or a residual expression, independently of the concrete result of partially evaluating this expression. Binding time analysis produces a safe approximation of this information.

For example, the specializer does not need the actual result of partially evaluating the test part of a conditional expression to decide whether it can solve the test and reduce the expression to one of its alternatives, or whether it needs to build a residual conditional expression. Again, this is important for self-application purposes since it makes it possible to avoid processing both parts of the specializer that handle reducing and rebuilding a conditional expression. This makes self-application faster and residual programs smaller.

1.3 An example: conditional expressions

As pointed out in the introduction, specialization occurs in a context. For example, all the sub-expressions of a static expression are evaluated. For another example, all the sub-expressions of a completely dynamic expression will be reproduced verbatim. For a last example, a conditional expression may be reduced if its test part is static, or rebuilt if it is dynamic. In the former case, the consequent and the alternative are specialized in the same context as the conditional expression.

Next section describes how we can build action trees by using a non-standard interpretation of the present semantics. These action trees repre-

sent further exploitation of the results of binding time analysis.

2 Semantics-Based Derivation of Action Trees

This section describes how to derive action trees from the specialization semantics. We have factorized it and defined a non-standard interpretation [Jones & Nielsen 89] generating action trees. We have instantiated the domain of residual programs to be the domain of action trees, and have provided a set of combinators to generate these trees. The result is a semantics-directed specification for deducing action trees from source programs and binding time trees.

2.1 The set of actions

In the factorization, there are two combinators for each syntactic construct. One captures the action of reducing the syntactic construct; the other represents the action of rebuilding the syntactic construct. Let us call these two actions respectively *Rd* and *Rb*.

This set of actions may be refined. Indeed *Rd* and *Rb* include respectively purely static expressions and purely dynamic expressions. Therefore, using this set, a specializer would interpret a purely static expression symbolically and would rebuild a purely dynamic expression. As an optimization, we enrich the set of actions with *Ev*, that denotes a purely static expression, and *Id*, that denotes a purely dynamic expression. Because the actions are more precise, the specializer may perform more accurate treatments, and thus be more efficient.

2.2 The action trees interpretation

Action trees are built using a non-standard interpretation of the factorized semantics: instead of residual expressions, we want to build action trees isomorphic to the binding time trees. This derivation is not detailed in this extended abstract. Instead, we give a set of simplified rules that capture the interpretation, in figure 6 and in appendix.

These rules are simplified because they only specify the action associated to each expression. Also, they do not account for the accumulation of trees.

Constant expressions are either evaluated or left identical according to their binding time value. Static and dynamic identifiers denote an action.

$$\rho \vdash [\![\texttt{(*cst C)}]\!]\, s : Ev \qquad \rho \vdash [\![\texttt{(*cst C)}]\!]\, d : Id$$

$$\rho \vdash [\![\texttt{(*ide I)}]\!]\, s : \rho\texttt{I} \qquad \rho \vdash [\![\texttt{(*ide I)}]\!]\, d : \rho\texttt{I}$$

$$\frac{\rho \vdash [\![\texttt{E-1}]\!]\, s : Ev \qquad \rho \vdash [\![\texttt{E-2}]\!]\, b_2 : Ev \qquad \rho \vdash [\![\texttt{E-3}]\!]\, b_3 : Ev}{\rho \vdash [\![\texttt{(*if E-1 E-2 E-3)}]\!]\, (s, b_2, b_3) : Ev}$$

$$\frac{\rho \vdash [\![\texttt{E-1}]\!]\, s : a_1 \qquad \rho \vdash [\![\texttt{E-2}]\!]\, b_2 : a_2 \qquad \rho \vdash [\![\texttt{E-3}]\!]\, b_3 : a_3}{\rho \vdash [\![\texttt{(*if E-1 E-2 E-3)}]\!]\, (s, b_2, b_3) : Rd}$$

$$\frac{\rho \vdash [\![\texttt{E-1}]\!]\, d : Id \qquad \rho \vdash [\![\texttt{E-2}]\!]\, b_2 : Id \qquad \rho \vdash [\![\texttt{E-3}]\!]\, b_3 : Id}{\rho \vdash [\![\texttt{(*if E-1 E-2 E-3)}]\!]\, (d, b_2, b_3) : Id}$$

$$\frac{\rho \vdash [\![\texttt{E-1}]\!]\, d : a_1 \qquad \rho \vdash [\![\texttt{E-2}]\!]\, b_2 : a_2 \qquad \rho \vdash [\![\texttt{E-3}]\!]\, b_3 : a_3}{\rho \vdash [\![\texttt{(*if E-1 E-2 E-3)}]\!]\, (d, b_2, b_3) : Rb}$$

$$\frac{\rho \vdash [\![\texttt{E-1}]\!]\, s : Ev \quad \ldots \quad \rho \vdash [\![\texttt{E-m}]\!]\, s : Ev}{\rho \vdash [\![\texttt{(*app op E-1...E-m)}]\!]\, (s, \ldots, s) : Ev}$$

$$\frac{\rho \vdash [\![\texttt{E-1}]\!]\, b_1 : Id \quad \ldots \quad \rho \vdash [\![\texttt{E-m}]\!]\, b_m : Id}{\rho \vdash [\![\texttt{(*app op E-1...E-m)}]\!]\, (b_1, \ldots, b_m) : Id}$$

$$\frac{\rho \vdash [\![\texttt{E-1}]\!]\, b_1 : a_1 \quad \ldots \quad \rho \vdash [\![\texttt{E-m}]\!]\, b_m : a_m}{\rho \vdash [\![\texttt{(*app op E-1...E-m)}]\!]\, (b_1, \ldots, b_m) : Rb}$$

$$\frac{\rho \vdash [\![\texttt{E-1}]\!]\, b_1 : Ev \quad \ldots \quad \rho \vdash [\![\texttt{E-n}]\!]\, b_n : Ev}{\rho \vdash [\![\texttt{(*app up E-1...E-n)}]\!]\, (b_1, \ldots, b_n) : Ev}$$

$$\frac{\rho \vdash [\![\texttt{E-1}]\!]\, b_1 : a_1 \quad \ldots \quad \rho \vdash [\![\texttt{E-n}]\!]\, b_n : a_n}{\rho \vdash [\![\texttt{(*app up E-1...E-n)}]\!]\, (b_1, \ldots, b_n) : Rd}$$

$$\frac{\rho \vdash [\![\texttt{E-1}]\!]\, b_1 : Id \quad \ldots \quad \rho \vdash [\![\texttt{E-n}]\!]\, b_n : Id}{\rho \vdash [\![\texttt{(*app sp E-1...E-n)}]\!]\, (b_1, \ldots, b_n) : Id}$$

$$\frac{\rho \vdash [\![\texttt{E-1}]\!]\, b_1 : a_1 \quad \ldots \quad \rho \vdash [\![\texttt{E-n}]\!]\, b_n : a_n}{\rho \vdash [\![\texttt{(*app sp E-1...E-n)}]\!]\, (b_1, \ldots, b_n) : Rb}$$

Figure 6: Inference rules for deducing actions
In the rules for application, op, up, and sp denote the names of an operator, an unfoldable procedure, and a specialization point, respectively.

In general all syntactic constructs that would be reduced are either *Ev* if all their components are *Ev*, or *Rd*; and all syntactic constructs that would be rebuilt are either *Id* if all their components are *Id*, or *Rb*.

3 Specializing Programs using Action Trees

This section discusses the structure of a specializer driven by action trees. Our set of actions abstracts the treatment of a specializer processing binding time trees. Therefore this treatment will be part of the new specializer processing action trees. The major difference concerns the decision as to which action to apply – since this decision has already been taken. Thus specialization essentially amounts to dispatching on the action.

As a result, the specializer is structured in four parts, evaluating, reducing, rebuilding, or reproducing each syntactic construct. Next step is to introduce the combinators for *Ev* and *Id*-expressions.

4 Defining Combinators for Specialization

Two out of the four specialization contexts entail a treatment that is completely independent from specialization: in a context *Ev*, source expressions are merely evaluated; in a context *Id*, expressions are merely reproduced. As computer scientists we have better ways than symbolic interpretation to evaluate or to reproduce constant Scheme expressions:

- we can run the *Ev*-expressions using the underlying Scheme processor;
- we could invoke the identity function on *Id*-expressions.

Using the underlying Scheme processor is faster by an order of magnitude, since we can compile the *Ev*-expressions instead of interpret them symbolically. It is not the rôle of a partial evaluator to mimic an evaluator by symbolic interpretation.

Nothing can beat the identity function, speedwise. It is not the rôle either of a partial evaluator to mimic the identity function by symbolic interpretation.

These observations suggest to extract the expressions annotated with these actions and to transform them into *combinators*. The idea is that both

Ev- and *Id*-combinators can be considered as primitive operations by the specializer.

Extracting *Ev*- and *Id*-combinators from a source program using its action trees is straightforward. Free variables are collected and abstracted. The corresponding definitions are added as static operators (*i.e.*, primitive procedures) or as residual procedures in the residual program.

Once *Ev*- and *Id*-combinators have been extracted from a source program, what remains is the essence of this program with respect to its partial input. It is also the essence of specialization: reduction and reconstruction of source expressions, and their management.

Because *Ev*-expressions are encapsulated in combinators, they are now treated as a call to a primitive and executed by the underlying machine. Because *Id*-expressions are encapsulated in combinators, a call to an *Id*-combinator is simply frozen. As a consequence, the specializer does not have to treat actions *Ev* and *Id*. To avoid multiplying trivial combinators, the treatment of some syntactic constructs survives: this concerns constants, variables, and calls to an operator.

5 Assessments

This section investigates various consequences of having observed and introduced action trees in the process of specialization and combinators in source programs.

5.1 Source programs

Because source programs are smaller, once they are pruned, they are faster to specialize. It is hard to characterize the average improvement, but experience confirms that pruning *Ev*- and *Id*-subtrees increases the speed of specialization.

5.2 Self-application

Still the classical question may be raised: is this self-applicable? To this question there is an immediate answer: it is self-applicable *a fortiori* because compiling binding times goes beyond binding time analysis, by exploiting binding time information further and moving static computation away from

binding declarations, *e.g.*, let expressions or residual lambda-expressions. This potentially yields name clashes that are usually avoided by systematically renaming residual variables.

Unfortunately this renaming impedes the purely dynamic semantics of specialization, since it is carried even for completely dynamic subexpressions – where there just cannot be any name clash, and thus no variable need to be renamed.

This problem is addressed and solved in [Danvy 89]. The practical consequence concerns the abstract syntax of source programs, where formal parameters are now qualified by a tag, as in figure 2. Each tag expresses whether the corresponding binding should be unfolded or whether it should be left residual; and if it is to be residual, whether the variable needs to be renamed.

5.6 An instruction set for specialization

We conjecture that the combinators we succeeded to extract from a source program hint at the existence of an instruction set in an architecture for specialization. This could be the basis for an algebra of specialization. We are currently investigating this issue.

5.7 An experiment: the generation of a linear string matcher

In [Consel & Danvy 89], we illustrated how the Knuth, Morris & Pratt linear matching algorithm could be derived from a naive, quadratic matching program by binding times analysis, staging, and specialization. Running both versions of the specializer (*i.e.*, the binding time-based version and the action-based one) shows an order-of-magnitude improvement of specialization when the static string is repetitive. This is due to the fact that in one case, matching the pattern (statically) against itself is interpreted whereas it is compiled in the other case.

5.8 Generality and extensibility of the approach

Compiling binding times is a general strategy because it does not rely on particular binding time analyses: action trees are solely generated from binding time information. This approach is extensible because the set of actions is extensible: the actions reflect the specialization process. Traditionally,

the specialization process. Because self-application is only a particular case of specialisation, the technique applies to self-application as well.

5.3 Combinators Rd and Rb?

One could imagine collecting *Rd*- and *Rb*-combinators as well. This would lead to fragment the process of specialization in even smaller units: each reduction and reconstruction would be handled by dedicated combinators, and specialization would be reduced to coordinating and combining these operations.

However is it worth it? A dedicated specializer (*i.e.*, the result of specializing a regular, Mix-like, specializer with respect to a source program) precisely offers this. As a matter of fact, all the *Ev*-, *Rd*-, *Rb*-, and *Id*-combinators can be identified in the text of the dedicated specializer – under unfolded form.

This observation again stresses the problem of granularity in semantics-based program manipulation. Aiming at exploiting binding time information further, we have succeeded to implement a new binding time shift in a self-applicable partial evaluator, by reorganizing and simplifying the basic steps of specialization. Because this has been achieved through a semantics-based derivation, it is not surprising that the result is the same in the end. However, intensionally the development is significant because it has been achieved without resorting to self-application. Source programs have become smaller and faster to specialize.

5.4 Purely static and purely dynamic semantics

Extracting *Ev*- and *Id*-combinators amounts to defining the purely static semantics of the source program as well as its purely dynamic semantics. We are now exploring what are the extensional meaning and the computational counterpart of these in, *e.g.*, the interpretive specification of a programming language, and their impact in the corresponding compilers obtained by self-application [Consel & Danvy 90].

5.5 On the purely dynamic semantics of specialization

During specialization, dynamic identifiers denote residual expressions, where potentially free variables occur. Among rebuilt expressions there may be

when one wants to strenghten partial evaluation, *e.g.*, to handle higher order functions or partially static structures, both the binding time analysis and the specializer have to be extended. The binding time analysis has to collect new information for the additional elements. The specializer has to be modified to treat these additional elements according to the binding time information. In our approach, we can extract additional actions from the specializer for capturing the new aspects of the specialization process. This has successfully been done when extending Schism to handle higher order functions and partially static structures, as reported in [Consel 90].

6 Comparison with Related Work

6.1 Binding time analysis

Binding time analysis is commonly agreed to be the necessary tool for realistic self-application. The problem with it is that it is too general: from [Jones & Muchnick 78] to [Nielsen & Nielsen 88], there is more in binding time analysis than self-applicable partial evaluation.

In this paper we argue that the information collected by the binding time analysis (namely: the binding time trees) can be exploited further, and that this exploitation can be dedicated to the process of specialization. We introduce action trees as characterizing specialization more precisely. Action trees are built by interpreting the binding time trees, which is done once and for all.

6.2 Actions in MIX

In Mix, a program is annotated with specialization actions after the binding time analysis. A set of two actions is used: eliminable and residual [Sestoft 86]. They respectively denote an expression whose syntactic construct will be statically reduced and an expression whose syntactic construct will be residual. Because nothing distinguishes a purely static expression from a purely dynamic one, both are interpreted symbolically.

6.3 Semantics-based compiler generation

Due to its self-application properties, partial evaluation has interesting applications to semantics-directed compiler generation. The point is that self-

applying the specializer with respect to the interpretive specification of a programming language yields a residual program having the functionality of a compiler. Correspondingly, the intensional Curry program obtained by specializing the specializer with respect to itself has the functionality of a compiler generator. Because we extract combinators from source specifications automatically, our generated compilers extract combinators as well.

In Mitchell Wand's framework and Peter Mosses's Action Semantics, a great emphasis is put on extracting or designing combinators. In contrast, our set of combinators is automatically abstracted during partial evaluation according to the binding times of the source program. Not surprisingly, considering that the goal is to specialize programs, such combinators are not as general as those found in Action Semantics. However they add up to automatizing the derivation of compilers and machine architectures from interpretive specifications.

Conclusions and Issues

Compiling binding times in a self-applicable partial evaluator goes beyond the effect of self-application and contributes to improving it too.

Extracting combinators captures the purely static and purely dynamic semantics of source programs.

Combining both provides two basic items abstracting structures from semantic specifications: in the case of an interpreter, self-application provides the compiling algorithm, and our combinators provide the instruction set of the corresponding machines, both for the compilation and the runtime program. The *Ev*-combinators are the instruction set for the compiler, and the *Id*-combinators are a part of the dynamic semantics.

Background and Acknowledgements

Actions were coined in [Consel 89] and implemented in Schism, a self-applicable partial evaluator for a dialect of Scheme. Then the second author observed that action trees and the corresponding specializer could be derived using a non-standard interpretation of the binding time-based specializer. Action trees have been successfully extended to tackle higher order functions and partially static structures [Consel 90].

Thanks go to Karoline Malmkjær, Siau Cheng Khoo, and Carolyn Talcott for commenting earlier versions of this paper.

References

[Bondorf *et al.* 88] Anders Bondorf, Neil D. Jones, Torben Æ. Mogensen, Peter Sestoft: *Binding Time Analysis and the Taming of Self-Application*, to appear in TOPLAS, DIKU, University of Copenhagen, Denmark (1988)

[Consel & Danvy 89] Charles Consel, Olivier Danvy: *Partial Evaluation of Pattern Matching in Strings*, Information Processing Letters, Vol. 30, No 2 pp 79-86 (1989)

[Consel 89] Charles Consel: *Analyse de Programme, Evaluation Partielle, et Génération de Compilateurs*, PhD thesis, University of Paris VI, Paris, France (June 1989)

[Consel 90] Charles Consel: *Higher Order Partial Evaluation with Data Structures*, Working paper, Computer Science Department, Yale University, New Haven, Connecticut (January 1990)

[Consel & Danvy 90] Charles Consel, Olivier Danvy: *Static and Dynamic Semantics Processing*, Technical Report 761, Computer Science Department, Yale University, New Haven, Connecticut (November 1989)

[Danvy 89] Olivier Danvy: *Avoiding Name Clashes during Self-Applicable Partial Evaluation*, Working paper, Computer Science Department, Indiana University, Bloomington, Indiana (fall 1989)

[Jones & Muchnick 78] Neil D. Jones, Steven S. Muchnick: *TEMPO: A Unified Treatment of Binding Time Parameter Passing Concepts in Programming Languages*, G. Goos & J. Hartmanis (eds.), Lecture Notes in Computer Science No 66, Springer-Verlag (1978)

[Jones *et al.* 89] Neil D. Jones, Peter Sestoft, Harald Søndergaard: *MIX: a Self-Applicable Partial Evaluator for Experiments in Compiler Generation*, Vol. 2, No 1 pp 9-50 of the International Journal LISP and Symbolic Computation (1989)

[Jones & Nielsen 89] Neil D. Jones, Flemming Nielsen: *Abstract Interpretation: a Semantics-Based Tool for Program Analysis*, to appear in the Handbook of Logic and Computer Science, University of Copenhagen and Aarhus University, Denmark (1989)

[Nielsen & Nielsen 88] Flemming Nielsen, Hanne R. Nielsen: *Automatic Binding Time Analysis for a Typed Lambda-Calculus*, proceedings of the ACM Symposium on Principles of Programming Languages pp 98-106 (1988)

[Rees & Clinger 86] Jonathan Rees, William Clinger (eds.): *Revised^3 Report on the Algorithmic Language Scheme*, Sigplan Notices, Vol. 21, No 12 pp 37-79 (December 1986)

[Sestoft 86] Peter Sestoft: *The Structure of a Self-Applicable Partial Evaluator*, pp 236-256 of *Programs as Data Objects*, Harald Ganzinger and Neil D. Jones (eds.), Lecture Notes in Computer Science No 217, Springer-Verlag (1986)

[Wand 82] Mitchell Wand: *Semantics-Directed Machine Architecture*, proceedings of the ACM Symposium on Principles of Programming Languages pp 234-241 (1982)

A Inference rules for deducing actions (continued)

$$\frac{\rho \vdash [\![\texttt{E-1}]\!]\, b_1 : Ev \;\ldots\; \rho \vdash [\![\texttt{E-k}]\!]\, b_k : Ev \qquad [\texttt{I-1} \mapsto Ev, \ldots, \texttt{I-k} \mapsto Ev]\rho \vdash [\![\texttt{E}]\!]\, b : Ev}{\rho \vdash [\![\texttt{(*let [u...] (I-1...) (E-1...) E)}]\!]\,((b_1, \ldots, b_k), b) : Ev}$$

$$\frac{\rho \vdash [\![\texttt{E-1}]\!]\, b_1 : a_1 \;\ldots\; \rho \vdash [\![\texttt{E-k}]\!]\, b_k : a_k \qquad [\texttt{I-1} \mapsto a_1, \ldots, \texttt{I-k} \mapsto a_k]\rho \vdash [\![\texttt{E}]\!]\, b : a}{\rho \vdash [\![\texttt{(*let [u...] (I-1...) (E-1...) E)}]\!]\,((b_1, \ldots, b_k), b) : Rd}$$

$$\frac{\rho \vdash [\![\texttt{E-1}]\!]\, b_1 : Id \;\ldots\; \rho \vdash [\![\texttt{E-k}]\!]\, b_k : Id \qquad [\texttt{I-1} \mapsto Id, \ldots, \texttt{I-k} \mapsto Id]\rho \vdash [\![\texttt{E}]\!]\, b : Id}{\rho \vdash [\![\texttt{(*let [i...] (I-1...) (E-1...) E)}]\!]\,((b_1, \ldots, b_k), b) : Id}$$

$$\frac{\begin{array}{c}\ldots \rho \vdash [\![\texttt{E-x}]\!]\, b_x : a_x \;\ldots \rho \vdash [\![\texttt{E-y}]\!]\, b_y : a_y \;\ldots \rho \vdash [\![\texttt{E-z}]\!]\, b_z : a_z \;\ldots \\ [\ldots, \texttt{I-x} \mapsto a_x, \ldots, \texttt{I-y} \mapsto a_y, \ldots, \texttt{I-z} \mapsto a_z, \ldots]\rho \vdash [\![\texttt{E}]\!]\, b : a\end{array}}{\begin{array}{c}\rho \vdash [\![\texttt{(*let [...u...i...r...] (...I-x...I-y...I-z...) (...E-x...E-y...E-z...) E)}]\!], \\ ((b_1, \ldots, b_x, \ldots, b_y, \ldots, b_z, \ldots, b_k), b) : Rb\end{array}}$$