

For a Better Support of Static Data Flow ^{*}

Charles Consel and Olivier Danvy

Yale University ^{**} and Kansas State University ^{***}

Abstract. This paper identifies and solves a class of problems that arise in binding time analysis and more generally in partial evaluation of programs: the approximation and loss of static information due to dynamic expressions with static subexpressions. Solving this class of problems yields substantial binding time improvements and thus dramatically better results not only in the case of partial evaluation but also for static analyses of programs — this last point actually is related to a theoretical result obtained by Nielson. Our work can also be interpreted as providing a solution to the problem of conditionally static data, the dual of partially static data.

We point out which changes in the control flow of a source program may improve its static data flow. Unfortunately they require one to iterate earlier phases of partial evaluation. We show how these changes are subsumed by transforming the source program into continuation-passing style (CPS). The transformed programs get specialized more tightly by a higher-order partial evaluator, without iteration. As a consequence, static values get frozen according to the specialization strategy and not due to the structure of the source programs.

Our approach makes it possible to get better results without changing our partial evaluator, by using its higher-order capabilities more thoroughly. By construction, transforming source programs into CPS makes it yield better results, even in the particular case of self-application. New problems can even be tackled such as static deforestation by partial evaluation, specialization of contexts, and conditionally static data.

This development concerns applicative order, side-effect free functional languages because we consider existing self-applicable partial evaluators. We conjecture a similar improvement for lazy functional languages, based on the normal order CPS transformation.

Keywords: Partial evaluation, binding times, continuation-passing style, Scheme, compilation of pattern matching, deforestation, partially static data, conditionally static data.

^{*} Proceedings of FPCA '91, John Hughes (ed.), LNCS 523, pp. 496–519.

^{**} Department of Computer Science, Yale University, P.O. Box 2158, New Haven, CT 06520, USA. This research was supported by DARPA under grant N00014-88-k-0573. Part of it was carried out while visiting Kansas State University in 1990. E-mail: consel@cs.yale.edu

^{***} Department of Computing and Information Sciences, Kansas State University, Manhattan, KS 66506, USA. Part of this work was carried out while visiting Yale University in 1990. E-mail: danvy@cis.ksu.edu

1 Introduction

Partial evaluation is a program transformation technique that aims at specializing programs with respect to part of their input. By definition, running the residual program on the remaining input yields the same result as running the source program on the complete input if they both terminate.

A partial evaluator reduces expressions in the source program as much as allowed by the available input, and reconstructs irreducible expressions, *i.e.*, expressions whose reduction depends on the unavailable parts of the input. Also, depending on the partial evaluation strategy and to ensure termination of the specialization process, an expression may be reconstructed even though it depends only on the available input; this issue goes beyond the scope of this paper but it is addressed in published work on self-applicable partial evaluation [26, 6, 3].

As initiated in the MIX project [25], the static and dynamic semantics of partial evaluation are best separated into two phases. The first phase — binding time analysis (BTA) — determines (a safe approximation of) the binding times of each source expression in a program, for a given division (known/unknown) of its input. The second phase — specialization — simply follows the binding time information: it reduces static expressions and reconstructs the other expressions.

Much of the work on partial evaluation has been devoted to improving the quality of residual programs. Indeed, partial evaluation occurs only once, whereas residual programs may run many times. In essence, this line of work aims at keeping track of the static parts of the data flow as accurately as possible, since the more static data are preserved throughout partial evaluation, the more source expressions get evaluated and therefore the less need to be executed in the residual program. However, this strategy does not address control structures whose specialization may freeze some static values as well as the computations that depend on these values.

As a simple example, let us consider the following applicative order expression

$$(\text{if } \underline{e_1} \text{ then } \overline{e_2} \text{ else } \overline{e_3}) + \overline{e_4}$$

where, for clarity, sub-expressions are annotated *à la* Nielson and Nielson. That is, an expression that is bound statically is overlined, otherwise it is bound dynamically and it is underlined.

Because the test expression e_1 cannot be evaluated at partial evaluation time, the conditional expression will be reconstructed. Therefore, the static values resulting from e_2 and e_3 will be frozen in the residual conditional expression although, outside, the second operand of the addition is static. As a consequence, the addition gets delayed until run time, despite the early binding time of both its operands.

Such losses of compile time values arise in many applications. For instance, in definitional interpreters, type checking operations get frozen; in pattern matchers, static knowledge about substitutions gets ignored.

Probably this problem can be solved by introducing more complicated mechanisms in the specialization process. We do not want to pursue this direction

because the specialization technique described above is simple and yet effective enough to yield, *e.g.*, non-trivial self-applicable partial evaluators.

Therefore, we propose to keep the same partial evaluator but to transform the control flow of the source program to prevent static values from being frozen. Such a meaning-preserving transformation amounts to reordering threads of execution in the source program to make use of the compile time values at compile time. Getting back to the example above, such a transformation would result in distributing the context (that is, the pending computations) in the conditional branches, thereby making the two instances of the addition executable at partial evaluation time.

$$\underline{\text{if } e_1 \text{ then } (\overline{e_2} \mp \overline{e_4}) \text{ else } (\overline{e_3} \mp \overline{e_4})}$$

Generalizing this approach based on reordering threads of execution is expensive since it requires one to iterate earlier phases of partial evaluation (*cf.* Section 3.) On the other hand, all threads could be reordered, thereby alleviating the need for iteration.

Regarding the extent of this radical transformation for a functional program, a more general version is actually well-known: it consists of translating the source program from direct style into continuation-passing style (CPS). From the point of view of partial evaluation, CPS programs appear to have numerous interesting properties: A CPS program is tail-recursive, thus, values are always passed forward; the resulting control flow improves on preserving the static parts of data flow with respect to the original source program. Further, because the CPS transformation essentially brings producers and consumers of intermediary data together, it becomes possible, at partial evaluation time, to eliminate the production and consumption of these data. Section 5 illustrates this strategy with a new approach to the deforestation problem [46]. Another novel aspect of this approach is the polyvariant specialization of contexts: each continuation is specialized with respect to different possible values.

Overview

The rest of this paper is organized as follows. Section 2 illustrates and characterizes the problem. Section 3 describes a solution based on a preliminary transformation of the source program into CPS and illustrates its effectiveness. Section 4 reviews the consequences of CPS for partial evaluation. Section 5 illustrates how partial evaluation can contribute to solving the deforestation problem and how contexts can be multiply specialized. Section 6 recasts our work as solving the problem of conditionally static data, the dual of partially static data. Finally, after a comparison with related work, our approach is put into perspective.

2 The Problem

This section illustrates the very common situation where the control flow in a source program hinders its static data flow. Then, we characterize the situations where the problem occurs by introducing a notion of context.

(defineType Pattern	(defineType Result	(defineType Substitution
(PatCst c)	(Unit)	(EmptySubst)
(PatVar v)	(Subst s))	(ExtendSubst v d s))
(PatSeq l))		

Fig. 1. Schism data types for pattern matching in lists.

Our source and residual programs are expressed in Scheme [39]. We are using Schism, a self-applicable, binding time-based partial evaluator for pure Scheme programs extended with user-defined data types [8]. Schism handles higher order functions as well as partially static data.

2.1 An example: linear pattern matching in lists

Pattern matching is a typical problem where partial evaluation applies well [2]. The idea is to compile a pattern by specializing a pattern matcher with respect to a pattern. The function computed by a compiled pattern expects a datum and attempts to match it. We consider pattern matching in pure Lisp lists. A pattern either declares a constant, introduces a variable, or specifies a sequence of (sub)patterns. Matching a pattern against a datum yields either an indication of failure, represented by the one-point type *Unit*, or a substitution. A substitution associates variables with values.

Figure 1 displays the data types of the pattern matcher. Figure 2 displays the executable specification of the pattern matcher, written with an accumulator holding the substitutions so far. It takes a pattern and a datum, and returns *Unit* or a list of substitutions. A constant pattern matches a datum if the constant equals the datum. A variable pattern yields the substitution of the datum for the variable. A sequence matches the corresponding list of data, recursively.

However specializing this program with respect to a pattern does not produce satisfactory compiled patterns (*cf.* Figure 3), for the following reasons.

The program is specialized with respect to a pattern: its first argument is available at partial evaluation time whereas its second argument will only be available at run time. Correspondingly, the syntactic analysis of the pattern will be performed at partial evaluation time whereas any computation involving the data will be performed at run time.

Matching a pattern against a datum either yields a substitution or *Unit*, depending on whether the match succeeds or fails. At partial evaluation time, the result of a match cannot be determined since the datum is not known until run time. As a consequence, the domain *Result* is dynamic. Because the pattern matcher is driven by the results of the matches to continue the computation, its control flow is dynamic as well. This is captured in the following binding time signatures:

```

;;; Pattern * Data -> Result
(define (main p d)
  (match p d (EmptySubst)))

;;; Pattern * Data * Substitution -> Result
(define (match p d s)
  (caseType p
    [(PatCst c) (if (equal? c d) (Subst s) (Unit))]
    [(PatVar v) (Subst (ExtendSubst v d s))]
    [(PatSeq l) (match-Seq l d s)]))

;;; List(Pattern) * Data * Substitution -> Result
(define (match-Seq l d s)
  (if (null? l)
    (if (null? d) (Subst s) (Unit))
    (if (null? d)
      (Unit)
      (caseType (match (hd l) (hd d) s)
        [(Unit) (Unit)]
        [(Subst s) (match-Seq (tl l) (tl d) s)]))))

```

Fig. 2. Linear pattern matching in lists, direct style with accumulator.

$$\begin{aligned}
\text{match} &: \overline{\text{Pattern}} \times \underline{\text{Data}} \times \widetilde{\text{Subst}} \Rightarrow \underline{\text{Result}} \\
\text{match-Seq} &: \overline{\text{List(Pattern)}} \times \underline{\text{Data}} \times \widetilde{\text{Subst}} \Rightarrow \underline{\text{Result}}
\end{aligned}$$

where static (partial evaluation time) domains are overlined, dynamic (run time) domains are underlined, and partially static domains are marked with a tilde. Here are the corresponding domains:

$$\begin{aligned}
\widetilde{\text{Subst}} &= \overline{\text{EmptySubst}} \mp (\overline{\text{Variable}} \times \underline{\text{Data}} \times \widetilde{\text{Subst}}) \\
\underline{\text{Result}} &= \overline{\text{Unit}} \pm \widetilde{\text{Subst}}
\end{aligned}$$

Because domain *Result* is dynamic, it is impossible to distinguish between *Unit* and a substitution statically. Therefore, residual programs such as the one in Figure 3 construct and decode intermediary results explicitly. The following section characterizes this pathological situation.

2.2 The problem of dynamic compound expressions

This section characterizes precisely when static subexpressions yield static values that get frozen in a dynamic expression, out of reach for a potential consumer. We outline the BNF of a Scheme program and introduce a notion of context.

```

;;; for all d, (main0 d) == (main '(PatSeq ((PatVar x) (PatCst 3))) d)

;;; Data -> Result
(define (main0 d)
  (if (null? d)
      (Unit)
      (caseType (Subst (ExtendSubst 'x (hd d) (EmptySubst)))
                [(Unit) (Unit)]
                [(Subst s)
                 (let ([d2 (tl d)])
                   (if (null? d2)
                       (Unit)
                       (caseType (if (equal? 3 (hd d2))
                                     (Subst s)
                                     (Unit))
                                 [(Unit) (Unit)]
                                 [(Subst s) (if (null? (tl d2))
                                               (Subst s)
                                               (Unit))])))])))

```

Fig. 3. Specialized version of the direct style pattern matcher w.r.t. a pattern.

This code was pretty-printed by hand. It is equivalent to the output of Schism. Obviously this program can be evaluated statically further, essentially by reducing the case expressions and propagating the corresponding information across the other conditional expressions. Nevertheless the specializer did not perform these simplifications, based on the too conservative information accumulated by the binding time analysis.

We formalize this notion with a BNF definition. Distinguishing between static, dynamic, and critical contexts, we pinpoint the possible loss of static information as a consequence of both the structure of a source program and the specialization strategy.

The syntactic categories of Scheme are predefined constants, identifiers, primitive calls, conditional expressions, applications, and abstractions. For simplicity, we leave blocks out of the discussion, since the **let** construct is a syntactic sugar for β -redexes and nested **letrec** expressions can be eliminated by lambda-lifting [24], yielding a set of mutually recursive functions as in Figure 2.

$$\begin{aligned}
 e ::= & \mathbf{cst}(c) \mid \mathbf{ide}(i) \mid \mathbf{cond}(e_1, e_2, e_3) \mid \\
 & \mathbf{prim}(op, (e_1, \dots, e_m)) \mid \mathbf{lam}((i_1, \dots, i_n), e) \mid \\
 & \mathbf{app}(e_0, (e_1, \dots, e_n))
 \end{aligned}$$

Specialization rules are simple: expressions either are reduced or they are reconstructed. Reducing a conditional expression depends on whether its test is

static or dynamic. The precise treatment of primitive operations depends on how partially static structures are handled. The decision to unfold or to residualize function calls is taken independently by the specializer. Static arguments are transmitted to the callee.

Such program transformations can freeze static values when a compound expression gets reconstructed. This possibility can be characterized syntactically, since static expressions yield static values. The following two definitions capture the relationship between an expression and a sub-expression.

Definition 1. The *context* of an expression is its immediate ancestor in the abstract syntax tree.

For example, in the expression $\mathbf{cond}(e_1, e_2, e_3)$, the context of e_1 is $\mathbf{cond}([\], e_2, e_3)$.

We can derive the following context constructors from the BNF of Scheme expressions. With a slight abuse of notation, let us specify them using a BNF format:

$$E[\] ::= \mathbf{cond}([\], e_2, e_3) \mid \mathbf{cond}(e_1, [\], e_3) \mid \mathbf{cond}(e_1, e_2, [\]) \mid \\ \mathbf{prim}(\text{op}, (e_1, \dots, [\], \dots, e_m)) \mid \mathbf{lam}((i_1, \dots, i_n), [\]) \mid \\ \mathbf{app}([\], (e_1, \dots, e_n)) \mid \mathbf{app}(e_0, (e_1, \dots, [\], \dots, e_n))$$

where the e_i denote Scheme expressions. In the following, we will refer to this as the “BNF of contexts.”

Definition 2. A *compound* expression $E[e]$ is obtained by filling a context $E[\]$ with an expression e .

For example, given the expressions e_1 , e_2 , and e_3 , the compound expression $\mathbf{cond}(e_1, e_2, e_3)$ can be built out of e_1 and the context $\mathbf{cond}([\], e_2, e_3)$; e_2 and $\mathbf{cond}(e_1, [\], e_3)$; and, e_3 and $\mathbf{cond}(e_1, e_2, [\])$.

Contexts make it possible to relate siblings in an abstract syntax tree. The following definition generalizes binding times to contexts.

Definition 3. The context $E[\]$ of an expression e is *static* if the corresponding compound expression $E[e]$ is reduced during specialization, and *dynamic* if it is reconstructed during specialization.

For example, if e_1 is bound statically, the context $\mathbf{cond}(e_1, [\], e_3)$ is static since given the expression e_2 , the corresponding compound expression $\mathbf{cond}(e_1, e_2, e_3)$ is reduced independently of what happens to e_2 and e_3 . Correspondingly, if e_1 is bound dynamically, the context $\mathbf{cond}(e_1, [\], e_3)$ is dynamic.

As can be noticed, a static context does not necessarily yield a static value. For example, a conditional expression whose test is static and branches are dynamic forms a static context with respect to its test. However, reducing this conditional expression yields a dynamic value.

Let us now connect the binding time of an expression with the binding time of its context.

Definition 4. The context $E[]$ of an expression e is *critical* if the corresponding compound expression $E[e]$ is reduced or reconstructed depending on whether e is static or dynamic. We then say that e *determines* $E[]$.

For example, if e_2 is bound statically, the context $\mathbf{prim}(+, ([], e_2))$ of the expression e (*i.e.*, the corresponding compound expression is $\mathbf{prim}(+, (e, e_2))$) is critical because it is reduced or reconstructed depending on whether e is static or dynamic. For another example, the context $\mathbf{cond}(e_1, [], e_3)$ is not critical.

Now, following Strachey, we can view each expression in an abstract syntax tree as producing a value, and we can identify how these expressible values are *consumed* by their context: as a test in a conditional expression; as an operand in a primitive operation; and as a function in an application. The other expressions only *transmit* values: as a conditional branch; and as an argument to a function.

We can even view the path between an expression and its consumer in the abstract syntax tree as a construction where the constructors are contexts. For example, the path between e_3 and the consumer $\mathbf{prim}(+, ([], e_4))$ in the expression $\mathbf{prim}(+, (\mathbf{cond}(e_1, e_2, e_3), e_4))$ is a construction where the constructors are $\mathbf{cond}(e_1, e_2, [])$ and $\mathbf{prim}(+, ([], e_4))$.

Definition 5. A context is *critical in a path* if it is critical and occurs in this path.

In the expression $\mathbf{prim}(+, (\mathbf{cond}(e_1, e_2, e_3), e_4))$, if e_4 is bound statically, the context $\mathbf{cond}([], e_2, e_3)$ is critical in the path between $\mathbf{prim}(+, (\mathbf{cond}(e_1, e_2, e_3), e_4))$ and e_1 . On the other hand, the context $\mathbf{cond}(e_1, [], e_3)$ is not critical in the path between this expression and e_2 .

Property 1 *An expression that determines its critical context can itself be compound.*

Proof: *cf.* BNF of contexts. □

In other terms, several contexts can be critical on the same path. An expression e may not determine its context $E[]$, but the corresponding compound expression $E[e]$ may determine its own context. Assuming $E[]$ to be reduced statically, e actually may determine the context of $E[e]$. Looking back at the previous example, e_2 and e_3 actually determine $\mathbf{prim}(+, ([], e_4))$

This syntactic non-locality of semantic dependence makes it possible to have interferences in the static data flow, as captured in the following property.

Property 2 *A static expression can be separated from its consumer (a test, a primitive operation, or an application) by a dynamic context which is not critical in the path between the expression and the consumer.*

Proof: *cf.* BNF of contexts (or more simply, the previous example.) □

A static expression that is separated from its consumer by a dynamic context gets frozen.

Consequence 1 *If a dynamic expression determines a critical context, all the siblings of this expression in the abstract syntax tree get frozen.*

This was the point of the example in the introduction:

$$\mathbf{prim}(+, (\mathbf{cond}(\underline{e_1}, \overline{e_2}, \overline{e_3}), \overline{e_4}))$$

Because of $\underline{e_1}$, $\overline{e_2}$ and $\overline{e_3}$ yield static values that are frozen. Similarly, $\overline{e_4}$ yields a value that is frozen.

In a nutshell, contexts are critical due to two distinct reasons: the specialization strategy and the structure of the source program. The former is understandable since partial evaluation subsumes constant propagation. The latter makes it clear why some programs “specialize better” than others.

3 A Solution

The problems listed in Section 2 have a common pattern: static values get frozen in residual expressions, out of reach for their consumers at partial evaluation time. Obviously, these static values and their consumers somehow should be put together despite the dynamic contexts in between. This could be achieved at different stages of partial evaluation, by changing either preprocessing (if there is any), or specialization, or postprocessing. In essence, such a strategy would amount to

1. identifying frozen static values and their consumers in an expression,
2. performing necessary transformations to bring them together, and
3. iterating the earlier phases of partial evaluation.

Static values get frozen in dynamic compound expressions because they cannot be reached by their consumers. Hence the idea, in step 2, is to distribute static contexts across dynamic compound expressions. This transformation makes sense because the present problem arises from the very structure of source programs, not from their specialization. However, the classification of static and dynamic constructs is determined by the structure of the program. Changing this structure requires one to re-analyze the binding times of the program, which in turn, may trigger new distributions of static contexts inside dynamic compound expressions, requiring a new binding time analysis, and so on — hence step 3. The number of iterations required depends on a given application.

A radical strategy would amount to bringing all potential values and corresponding consumers together, regardless of the binding times of these values. This would remove the need for iterating because repeated transformations are necessary only to propagate newly available values, which is done once and for all by the radical transformation that distribute *all* contexts inside *all* dynamic expressions, even function calls. This is an extreme approach, because presumably not all contexts need to be distributed; however, it yields definite results: further modifications of the program or iterations of the binding time analysis are

not required. Since this radical transformation occurs before partial evaluation, it improves the static data flow of a program without any modification of the partial evaluator.

Let us first analyze how the radical transformation is naturally achieved by CPS transformation. Then, we will illustrate its effect on the pattern matching example, and finally take a critical look at the whole process.

3.1 Continuation-passing style

Transforming a program into CPS essentially amounts to representing each context as a lambda-expression, distributing this “continuation” across all control structures, and in particular passing it as an extra argument to each function of the original program [38, 15]. As a net effect, CPS programs are tail-recursive. Further, when a call is specialized, static values still are passed to the callee to specialize it — including what is static in the continuation.

Here is a BNF of CPS terms.

$$\begin{aligned} e ::= & \mathbf{cond}(t, e_2, e_3) \mid \mathbf{app}(t_0, (t_1, \dots, t_n, t_{n+1})) \\ t ::= & \mathbf{cst}(c) \mid \mathbf{ide}(i) \mid \mathbf{prim}(op, (t_1, \dots, t_m)) \mid \\ & \mathbf{lam}((i_1, \dots, i_n, i_{n+1}), e) \mid \mathbf{lam}((i_1, \dots, i_n, i_{n+1}), t) \end{aligned}$$

A continuation has been added as an extra argument to functions. The terms t are “trivial” [40], *i.e.*, evaluating them cannot loop. In terms of “sub-problems” and “reductions” popular in the Scheme community [19], all the sub-problems amount to evaluating trivial expressions.

Following Section 2.2, let us derive the BNF of contexts for a CPS program.

$$\begin{aligned} E[] ::= & \mathbf{lam}((i_1, \dots, i_n, i_{n+1}), []) \mid \mathbf{cond}(t, [], e_3) \mid \mathbf{cond}(t, e_2, []) \\ T[] ::= & \mathbf{cond}([], e_2, e_3) \mid \\ & \mathbf{app}([], (t_1, \dots, t_n, t_{n+1})) \mid \mathbf{app}(t_0, (t_1, \dots, [], \dots, t_n, t_{n+1})) \mid \\ & \mathbf{prim}(op, (t_1, \dots, [], \dots, t_m)) \mid \mathbf{lam}((i_1, \dots, i_n, i_{n+1}), []) \end{aligned}$$

where the e_i and t_i are defined by the BNF above.

This BNF of contexts reveals a crucial property that parallels Property 1:

Property 3 *All the expressions that determine a critical context are trivial.*

Proof: by cases. □

Lemma 6. *Only trivial terms can be consumed.*

Proof: *cf.* BNF of CPS expressions. □

The following property parallels Property 2.

Property 4 *The only possible context which is not critical in the path between a static expression and its consumer is $\mathbf{app}(t_0, (t_1, \dots, [], \dots, t_n, t_{n+1}))$*

Proof: *cf.* BNF of contexts. □

The following consequences parallel Consequence 1.

Consequence 2 *There is no dynamic context in the path between a static expression and its consumer.*

No static value is frozen by a context on the path to its consumer since there is no compound expressions anymore but function calls. (Remember that by definition of the class of partial evaluators we are considering, static arguments are transmitted to the callee.)

Consequence 3 *A static value gets frozen because its consumer is a primitive operation and one of the siblings of the static value is dynamic.*

In other terms, if a static value has a static consumer, it will reach this consumer and be consumed. Otherwise a static value has a dynamic consumer; it will reach this consumer and be frozen.

Consequence 4 *The decision to reduce or to reconstruct a term only depends on the specialization strategy and not on the structure of the source program.*

To sum up, the problem described in Section 2, by construction, cannot occur for CPS terms. Since the CPS transformation is meaning-preserving, this motivates the transformation of source programs into CPS.

3.2 Example (continued): linear pattern matching in lists

Transforming the program of Figure 2 into CPS yields the program displayed in Figure 4 and the following domains, annotated as in Section 2.1:

$$\begin{aligned}\widetilde{Subst} &= \overline{EmptySubst} \mp (\overline{Variable} \times \underline{Data} \times \widetilde{Subst}) \\ \widetilde{Result} &= \overline{Unit} \mp \widetilde{Subst} \\ \underline{Answer} &= \overline{Unit} \pm \widetilde{Subst}\end{aligned}$$

The domain of results is now a static sum. Therefore whether a result is *Unit* or a substitution can be determined at partial evaluation time. As a consequence, all matching operations depending on the structure of patterns and on substitutions (syntactic analysis and propagation of intermediary substitutions) are performed at partial evaluation time. In fact, and as discovered in [14] and analyzed in Section 5, compiled patterns operate even better than when they are interpreted because all the structural tests occur first and the substitution is built only if the match succeeds, instead of incrementally, and potentially for nothing in case of mismatch. This is illustrated by Figure 5 that displays the result of specializing the transformed pattern matcher with respect to the same pattern as in Figure 3.

```

;;; Pattern * Data -> Answer
(define (main p d)
  (match p d (EmptySubst) (lambda (r) r)))

;;; Pattern * Data * Substitution * [Result -> Answer] -> Answer
(define (match p d s k)
  (caseType p
    [(PatCst c) (if (equal? c d) (k (Subst s)) (k (Unit)))]
    [(PatVar v) (k (Subst (ExtendSubst v d s)))]
    [(PatSeq l) (match-Seq l d s k)])

;;; List(Pattern) * Data * Substitution * [Result -> Answer] -> Answer
(define (match-Seq l d s k)
  (if (null? l)
      (if (null? d) (k (Subst s)) (k (Unit)))
      (if (null? d)
          (k (Unit))
          (match (hd l) (hd d) s (lambda (r)
                                   (caseType r
                                     [(Unit)
                                      (k (Unit))]
                                     [(Subst s)
                                      (match-Seq (tl l) (tl d) s k)])))))))

```

Fig. 4. Linear pattern matching in lists, continuation-passing style with accumulator.

This program is the CPS counterpart of the program of Figure 2. It was obtained automatically using the CPS transformation for λ_v -terms [15].

4 Appraisal

Binding time analysis [26] abstracts the specialization semantics. As an abstract interpretation, it is less accurate than the semantics it abstracts. In other words, since binding time analysis does not operate on concrete values, it needs to approximate to yield “safe” information. Thus some expressions may be classified to be run time when they are actually compile time and therefore, fewer computations are performed during specialization. However, regarding CPS expressions, the only conservative effects of the BTA concern the final value, which is dynamic anyway.

The point of the CPS transformation is to bring contexts to expressions across conditional expressions and function calls. The CPS transformation brings partial evaluators that include a (necessarily approximate) BTA closer to the accuracy of those that do not include an explicit BTA at preprocessing time. Yet CPS transformation keeps the partial evaluators with an offline BTA more effective than the partial evaluators with an online discrimination because of the pre-

```

;;; for all d, (main0 d) == (main '(PatSeq ((PatVar x) (PatCst 3))) d)

;;; Data -> Answer
(define (main0 d)
  (if (null? d)
      (Unit)
      (let ([d2 (tl d)])
        (cond
         [(null? d2) (Unit)]
         [(equal? 3 (hd d2))
          (if (null? (tl d2))
              (Subst (ExtendSubst 'x (hd d) (EmptySubst)))
              (Unit))]
         [else (Unit)]))))

```

Fig. 5. Specialized version of the CPS pattern matcher w.r.t. the same pattern as in Figure 3.

This dedicated program traverses its argument depth-first, iteratively. All the continuations of the source program depended on the structure of the pattern only and thus have been eliminated. There are no computation duplications because of the let expression. The substitution is built only if the match succeeds. Again, this code was pretty-printed by hand. It is equivalent to the output of Schism.

computations based on the static binding time information. (Obviously, the best of both worlds is obtained with a partial evaluator that does not trust the dynamic binding time information, and analyzes the actual binding time of “dynamic” values online.)

Transforming the source program into CPS is a conservative action: as a source to source transformation, it does not introduce any new syntactic form that would require some special treatment.

However, transformation into CPS introduces a series of higher order functions that need to be treated explicitly, whereas implicit contexts were treated without ado. Still nothing there should frighten a higher order partial evaluator. Better, given a polyvariant one, *i.e.*, a partial evaluator that can produce multiple specialized versions of a function, pre-transforming the source program into CPS makes it possible to produce multiple specialized versions of source contexts and to eliminate intermediary data. This point is illustrated in Section 5.

Regarding termination, CPS programs introduce a new component to take into account: the continuation. In some cases the continuation needs to be generalized to ensure termination of partial evaluation. To examine this issue let us compare termination of partial evaluation for direct style and CPS programs in a typical case. Consider a recursive function with an induction variable bound dynamically.

If this function is expressed in direct style, then, as described by Sestoft, for example [41], it should be specialized to delay recursion until run time (*i.e.*, when the value of the induction variable is available). If this function is expressed in CPS, then in addition to its specialization, the continuation has to be generalized. Indeed, the continuation may accumulate computations and thus cause infinite specialization. In other words, for CPS programs, not only is the control expressed by recursion, but it is also captured by the continuation. Thus, to freeze the control of a function at partial evaluation time, this function has to be specialized as well as generalized in its continuation argument.

In fact, this reasoning subsumes the traditional problems of accumulators as static values under dynamic control,⁴ since accumulators are nothing but concrete representation of continuations [48]. Their (static) initial value always has required to be generalized to ensure termination of partial evaluation.

Let us apply this reasoning to the pattern matcher presented previously. First, notice that the pattern is the induction variable; also, recall that this variable is assumed to be bound statically. In that case, recursive calls can be unfolded and continuations propagated. If the pattern matcher was invoked with both a dynamic pattern and a dynamic datum, then recursive calls would have to be suspended and continuations generalized.

This generalization strategy can be refined using Chin's approach for removing higher order functions [5]. The idea is to propagate a functional argument — in our case a continuation — if the corresponding parameter is *variable-only*. Essentially a parameter of a function has this property if the corresponding argument in each recursive call to this function is made up of only variables. Of course, this property trivially holds for all parameters of non-recursive functions.

Chin uses this simple syntactic property to ensure successful folding of recursive expressions during deforestation. In the context of partial evaluation of CPS programs, this property ensures that a continuation parameter will not cause infinite specialization because it is bound to finitely many functional values.

Finally, let us turn to self-application. CPS code has a very special property: it is completely tail-recursive. As a direct consequence, no value is ever returned but at the end. Thus if a static value gets frozen in a dynamic context, all outer contexts are dynamic. There is no outer critical context to be deprived of this static value. Hence the following proposition.

Proposition 7. *Let $\text{run PE } \langle \text{PE}, \langle p, _ \rangle \rangle$ denote the self-application of PE to p . If p is a CPS program, the inner instance of PE need not be transformed into CPS.*

Proof. The only reason to transform a program is to bring critical contexts to static values. Since p is a CPS program, specializing each of its expressions never

⁴ Consider a recursive function computing the length of a list, using an accumulator. Specialize it w.r.t. the (static) initial value of the accumulator. This will provoke either infinite unfolding or the generation of infinitely many specialized functions, one for each static value of the accumulator.

returns a value but at the end. Therefore there are no static contexts either, and thus no reason to transform *PE* into CPS.

This result parallels Shivers's static analyses that are in direct style because the analyzed programs are in CPS [42].

5 Applications

As is well-known, functional programs often make a momentary use of intermediary data.⁵ A great deal of optimizations have been developed to eliminate the construction of these data [4, 46]. The wasteful programs can be reformulated into others that do not use intermediary data. The two following sections detail the elimination of four kinds of intermediary data.

5.1 Eliminating partially static data structures

As a static semantics processor, a partial evaluator produces and consumes static data structures, whether they are recursive (static fixpoints are unfolded) or not. For example, in an interpreter, the static part of the environment, lives at compile time; as such, it is a partially static structure that only exists at partial evaluation time. For another example, static injection tags live at partial evaluation time as well.

5.2 Eliminating dynamic data structures

Due to the CPS transformation, tuples holding dynamic results (*i.e.*, non recursive and dynamic data structures) are now passed forward to the continuation. As statically constructed data, they are eliminated, *e.g.*, by raising the arity [33] of the continuation.

However partial evaluation falls short and cannot handle dynamic, recursive data structures because they are built under dynamic control. This conflicts with fixed specialization strategies.

5.3 Assessments

As reported in [14], CPS has considerable consequences both on source programs and on residual programs, in the case of pattern matching. In the source programs, (1) mismatching is solved statically as the application of the initial continuation; (2) the arity of the success continuation is raised by encoding the static tuples of results in the residual program; and (3) the static construction of static names and the static construction of dynamic values are separated. The new source program specializes very well with respect to any pattern. It yields tightly compiled patterns where (1) the list of names is built statically; (2) in the case of non-linear pattern matching, cross-references are solved statically; and (3) further, in

⁵ This is referred to as the functional facet in Peter Mosses's Action Semantics [35].

the residual program, the substitution is built only if matching succeeds, instead of incrementally as in the source pattern matcher.

Finally let us turn to compiling and generating a compiler for strongly typed programs. As revealed by experience, scope resolution, storage calculation, and static type checking get processed at compile time [11].

6 Conditionally Static Values

This section presents an alternative motivation and development for CPS as providing a better support of static data flow. Essentially, we mirror Mogensen's separation of binding times in source programs [34].

The expressions specifying the components of data structures often are bound at different times. For example [11], in an interpreter for statically typed programs, a denotable value may be represented with a pair holding the type tag and the actual value. Because the language is statically typed, the type tag depends only on the program, whereas the actual value depends on both the program and its input. The expressions specifying the type tag are bound early (at compile time) whereas the expressions specifying the actual value are bound later (at run time). As pairs holding the type tag and the actual value, denotable values form an example of partially static data.

Definition 8 Mogensen. A *partially static datum* is a static product of (partially) static or dynamic data.

In a partial evaluator that does not handle partially static data, any datum has the latest binding time of its components. In a partial evaluator that handles partially static data, a partially static datum has an early binding time, despite the late binding time of some of its components. This makes it possible to process partially static data at partial evaluation time.

Let us consider the dual of partially static data.

Definition 9. A *conditionally static datum* is a dynamic sum of (conditionally or partially) static or dynamic data.

N.B.: By the same token, we generalize partially static data to be static product of (partially or conditionally) static or dynamic data.

Conditionally static data results from dynamic conditional expressions. For example,

$$\mathbf{cond}(\underline{e_1}, \overline{e_2}, \overline{e_3}), \mathbf{cond}(\underline{e_1}, \underline{e_2}, \overline{e_3}), \text{ and } \mathbf{cond}(\underline{e_1}, \overline{e_2}, \underline{e_3})$$

are three expressions yielding conditionally static data at partial evaluation time.

Conditionally static data are approximated with the latest binding time of their components by any existing binding time analysis. The following typical rule reflects that the result of a conditional expression is dynamic whenever its test is bound dynamically:

$$\frac{\star \vdash e_1 : D}{\star \vdash \mathbf{cond}(e_1, e_2, e_3) : D}$$

How can we process conditionally static data in a partial evaluator?

Let us continue the parallel with partially static data. They are processed either with an explicit representation [32, 7] or by separating binding times, *i.e.*, by rewriting the source program to separate the static and dynamic expressions that build the components of partially static structures [34]. Similarly, conditionally static data could be processed either with an explicit representation of sum values, or by separating binding times. In both cases, we need to represent contexts.

Transforming the source program in CPS offers an elegant solution. The context is represented with a function: the continuation. Polyvariant specialization routinely performs the specialization of each continuation with respect to each potential component of a sum value, thereby separating binding times.

This puts a new requirement upon BTA-based partial evaluators: to have a polyvariant BTA. This is necessary for specializing continuations with respect to values that are bound at different times.

To overcome the limitation of partial evaluators with monovariant BTA, a solution consists of duplicating contexts during the CPS transformation. For example, translating the following direct style Scheme expression

```
(f (if (if x y z) 4 5))
```

into CPS yields the following voluminous term:

```
(lambda (g47)
  (if x
      (if y (f 4 g47) (f 5 g47))
      (if z (f 4 g47) (f 5 g47))))
```

where contexts (conditional expression and function application) have been duplicated.

Compiler writers usually frown upon duplicating contexts and favor translation schemas which yield CPS terms that are linear in size with respect to the original direct style term. For example, without duplicating contexts, the expression above yields:

```
(lambda (g52)
  (let ([k53 (lambda (v54)
              (let ([k55 (lambda (v56)
                          (f v56 g52))])
                (if v54 (k55 4) (k55 5)))]])
    (if x (k53 y) (k53 z))))
```

However, for partial evaluation, duplicating contexts is a practical way to introduce a restricted form of polyvariance without using a polyvariant BTA, just as

CPS transformation allows more thorough specialization while keeping the same specializer.

For completeness, let us list the four cases of building partially static and conditionally static data under static and under dynamic control:

	Product	Sum
Static context	(1)	(2)
Dynamic context	(3)	(4)

where by “static control” (resp. “dynamic control”) we mean that the expression yielding the partially or conditionally static data occurs in a static (resp. dynamic) context (*cf.* Definition 2).

(1) corresponds to partially static data and is illustrated by

$$\overline{\mathbf{hd}}(\overline{\mathbf{cons}}(e_1, e_2))$$

where **cons** builds a product value and the context **hd**([]) is reduced statically, yielding a value with the same binding time as e_1 .

(3) corresponds to a partially static datum whose construction is delayed until run time; it is illustrated by

$$\underline{\mathbf{equal}}(e_1, \overline{\mathbf{cons}}(\overline{e_2}, \underline{e_3}))$$

where **equal** denotes a primitive operation.

(4) corresponds to conditionally static data and is illustrated by

$$(\underline{\mathbf{if}} \underline{e_1} \underline{\mathbf{then}} \overline{e_2} \underline{\mathbf{else}} \underline{e_3}) + \overline{e_4}$$

(2) corresponds to the following irritant case:

$$(\underline{\mathbf{if}} \overline{e_1} \underline{\mathbf{then}} \underline{e_2} \underline{\mathbf{else}} \overline{e_3}) \pm \overline{e_4}$$

The binding time analysis cannot determine which expression e_2 or e_3 will be selected at specialization time. To ensure a safe approximation, the BTA assumes the result of the conditional expression to be dynamic, thereby freezing the siblings in the abstract syntax tree. Assuming e_1 to yield false statically, e_3 to yield 10, and e_4 to yield 20, the result of specializing this expression reads

$$10 + 20$$

which would not happen with a better support of static data flow such as the one offered by the CPS transformation.

Finally, let us conclude on partially *vs.* conditionally static data. Partially static data are built as products and essentially are treated using value-based strategies. Conditionally static data are built as sums and essentially are treated using continuation-based strategies. This is in pleasing relationship with Filinski’s work on duality in programming languages [17].

7 Related work

Very early, experience with MIX has shown how some programs may “specialize better” than others [16]. But we are not aware of any formalization of this phenomenon in the context of partial evaluation, as presented in Section 2.2, though Nielson’s work on data flow analysis in a denotational framework using abstract interpretation [36] relates to our endeavor. The most precise abstract interpretation is obtained by the Meet Over all Paths solution; in particular, Kam and Ullman have shown that in general it yields more precise results than the usual Maximal Fixed Point solution [29]. Considering the specification of an imperative language, Nielson has shown that the abstract interpretation of its direct style formulation usually leads to the Maximal Fixed Point solution whereas the abstract interpretation of a CPS formulation naturally leads to the Meet Over all Paths solution. This result actually forms a basis of our work because partial evaluation relates to the results of binding time analysis, which is an abstract interpretation; and because our pure version of Scheme (evaluation order notwithstanding) is close to the metalanguage of denotational semantics. Anyway continuation-passing terms are independent of their evaluation order [40].

Jørring and Scherlis’s staging transformations [28] closely relate to our approach. These source-to-source transformations aim at making a program more static. Static abstract syntax nodes are moved outwards in the abstract syntax tree. The transformation stops at conditional expressions. In contrast, transformation into CPS moves abstract syntax nodes inwards in the abstract syntax tree as this tree gets linearized. Also, CPS transformation does go through conditional expressions, thereby enabling considerably many more opportunities for actual specializations, but of course this is because we consider applicative order programs.

Mogensen’s paper on separating binding times [34] is commonly referred to as addressing binding time improvements of programs. In fact, Mogensen’s transformation uses the same binding time information as, say, a partial evaluator with partially static structures, to yield the same good results eventually. Separating binding times in a program does not improve its static data flow.

Nielson and Nielson aim at improving binding times of programs based on a static analysis identifying disagreement points [37]. For a class of higher-order programs, such information can be used to provide binding time improvements with respect to a fold/unfold strategy *à la* Burstall and Darlington.

This line of work is being pursued by Holst and Hughes who aim at applying Wadler’s theorems for free (based on results about parametric polymorphism) to obtain binding time improvements [22, 47]. Our point is simpler and it is implemented. We identify a class of binding time improvements and stay within the framework of partial evaluation, based on the usual congruence between direct style programs and their continuation-passing counterpart.

Drawing a parallel between partial evaluation of eager programs and lazy evaluation, Holst and Gomard propose simple transformations to improve the sharing properties of a lazy program [20, 21]. This makes it possible for the evaluation of such fully lazy programs to match the efficiency obtained by partial

evaluation of their eager counterpart. Holst and Gomard’s hand transformations address first-order programs and are subsumed by the usual and automatic higher-order CPS transformation. It would be interesting to investigate the effects of a more radical pre-transformation into normal order CPS, as illustrated in the present paper for applicative order programs.

Writing source programs iteratively was known to contribute to their efficient specialization (see, *e.g.*, [9, 6].) Earlier works by the authors on compiling patterns [14] or processing the static and dynamic semantics of Algol and Prolog [11, 12] have motivated continuation-passing style as generalizing iterative style to circumvent the loss of compile time values in a higher-order, recursive setting. The present work integrates this style in the process of partial evaluation, thereby making it possible to handle a broader class of source programs more accurately.

Exhibiting continuations and keeping backtracking under static control, as suggested by the second author in [14], have already proven to be effective in the area of pattern matching: in [27], Jørgensen applies this technique to the compilation of patterns as encountered in a case expression; in [43], Smith exhibits both success and failure continuations of a logic program to compile pattern matching.

So far, continuation-passing style has been used to compile Scheme and Standard ML programs [44, 30, 1], to derive compilers from interpreters [49], and for transforming programs [48]. Let us address these three points. Shivers [42] develops a number of flow analyses for CPS Scheme programs, but does not motivate CPS as something else but a convenient intermediate representation. Deriving compilers from interpreters is achieved generically using a self-applicable partial evaluator [18, 26, 11] and the present paper motivates why CPS is “the right thing” indeed. Transforming programs using an explicit representation of contexts does not seem to have been applied to eliminate intermediary data, as addressed in Section 5. Similarly, the specific problem of specializing contexts had not been identified so far, even though *e.g.*, Launchbury mentions its effects in [31, pages 107–108].

Our work is based on the CPS transformation, as described by Plotkin in [38]. Further developments on this transformation are reported by Filinski and the second author in [15]: essentially, the binding times in Plotkin’s CPS transformations can be improved, yielding more reasonable one-pass transformers; principally, the improvement is based on standardizing the sets of syntactic constructs yielded by a collecting interpretation.

Recent work by Weise and Ruf proposes “on the fly fixpoint iterations” to improve online partial evaluation [50] (this paper was brought to our attention after the present work was carried out.) We believe them to correspond to the iterations mentioned in Section 3. In the introduction of [50], Weise and Ruf also reject CPS as something incompatible with “termination extent strategies.” We did not meet this problem.

Finally, Turchin’s driving [45] is a program manipulation that also involves an explicit representation of the context. It is more general and powerful than partial evaluation in that a supercompiler actually can eliminate intermediary data structures, even if they are built under dynamic control, which partial evaluation

cannot do (*cf.* Section 5). Yet our work can be seen as providing an explicit representation of the context (as a function: the continuation), besides improving its binding times.

8 Conclusions and Issues

This article identifies the problem of programs that specialize better than others as a structural instead of a conjectural one. Essentially, we propose to change the control flow of the source programs to support their static data flow better, thereby ensuring more thorough specialization while keeping the same higher-order specializer. We circumvent the need for iterating partial evaluation by systematically transforming the source program into CPS. This development is compatible with self-application. It makes it possible to improve on earlier applications such as compiling and compiler generation [18, 26, 11] and also to tackle new problems such as static deforestation, the specialization of contexts, and conditionally static data.

CPS transformation is excessive in that it does not guarantee the improvement of binding time properties of a source function. In other terms, not all of a source program need to be expressed in CPS. For example, completely static expressions obviously need not be transformed. For another example, if we write the pattern matcher of Figure 2 without accumulator but by appending intermediary substitutions, the function appending these substitutions need not be transformed into CPS. In the general case, it is not clear how to obtain this “ideal” representation of the source program swiftly — in particular without iterating the binding time analysis. We are currently weighting the relative merits of the total CPS transformation (simplicity, automatism, and transparency) and of some partial CPS transformation (increased opportunity to extract more substantial static and dynamic combinators [10].) The latter appear to require a termination analysis to discriminate between Reynolds’s “serious” and “trivial” terms [40]. This could correspond to using strictness analysis for the normal order CPS transformation.

Residual programs in general are expressed in CPS. They can be mapped back into direct style, though not uniquely [38]. We are currently working on this issue.

In Section 4, we argue that our approach eliminates the approximations of the binding time analysis for conditional expressions. Indeed, CPS programs are tail-recursive: the context of a conditional expression is distributed over the branches. Beyond binding time analysis, this transformation appears applicable to other static analyses to circumvent similar approximations. This claim is supported by Nielson’s results in [36] (*cf.* Section 7.)

Finally, we are currently investigating the connections between forward and backward static analyses [13, 23] and CPS. This might lay off grounds to backwards partial evaluation of programs, *i.e.*, the specialization of programs with respect to their context of use, instead of the current mere forward methods.

As a last word, let us stress that this work contributes to the relief from writing source programs in a contrived way, “to make them specialize better.”

Acknowledgements

To Karoline Malmkjær, David Schmidt, and Andrzej Filinski. Thanks are also due to Siau Cheng Khoo, Olin Shivers, Don Smith, and the referee.

References

1. A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *ACM Symposium on Principles of Programming Languages*, pages 293–302, 1989.
2. D. Bjørner, A. P. Ershov, and N. D. Jones, editors. *Partial Evaluation and Mixed Computation*. North-Holland, 1988.
3. A. Bondorf and O. Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. DIKU Research Report 90/04, University of Copenhagen, Copenhagen, Denmark, 1990. To appear in *Science of Computer Programming*.
4. W. Burge. An optimizing technique for high level programming languages. Research Report RC 5834 (# 25271), IBM Thomas J. Watson Research Center, Yorktown Heights, New York, New York, 1976.
5. W. N. Chin. *Automatic Methods for Program Transformation*. PhD thesis, University of London, Imperial College of Science, Technology and Medicine, London, UK, 1990.
6. C. Consel. *Analyse de Programmes, Evaluation Partielle et Génération de Compilateurs*. PhD thesis, Université de Paris VI, Paris, France, June 1989.
7. C. Consel. Binding time analysis for higher order untyped functional languages. In *ACM Conference on Lisp and Functional Programming*, pages 264–272, 1990.
8. C. Consel. *The Schism Manual*. Yale University, New Haven, Connecticut, USA, 1990. Version 1.0.
9. C. Consel and O. Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, 1989.
10. 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.
11. C. Consel and O. Danvy. Static and dynamic semantics processing. In *ACM Symposium on Principles of Programming Languages*, pages 14–23, 1991.
12. C. Consel and S. C. Khoo. Semantics-directed generation of a Prolog compiler. In *PLILP'91, 3rd International Symposium on Programming Language Implementation and Logic Programming*, 1991. To appear.
13. P. Cousot. Semantic foundations of program analysis: Theory and applications. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.
14. O. Danvy. Semantics-directed compilation of non-linear patterns. *Information Processing Letters*, 37:315–322, March 1991.
15. O. Danvy and A. Filinski. Representing control, a study of the CPS transformation. Technical Report CIS-91-2, Kansas State University, Manhattan, Kansas, USA, 1991.
16. H. Dybkjær. Parsers and partial evaluation: An experiment. Diku student report 85-7-15, University of Copenhagen, Copenhagen, Denmark, 1985.

17. A. Filinski. Declarative continuations: An investigation of duality in programming language semantics. In D.H. Pitt et al., editors, *Category Theory and Computer Science*, number 389 in Lecture Notes in Computer Science, pages 224–249, Manchester, UK, September 1989.
18. Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls* 2, 5, pages 45–50, 1971.
19. C. Hanson. Efficient stack allocation for tail-recursive languages. In *ACM Conference on Lisp and Functional Programming*, pages 106–118, 1990.
20. C. K. Holst. Improving full laziness. In Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors, *Functional Programming, Glasgow 1990. Workshops in Computing*, pages 71–82. Springer-Verlag, August 1990.
21. C. K. Holst and C. K. Gomard. Partial evaluation is fuller laziness. In *Proceedings of the first ACM SIGPLAN and IFIP Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, New Haven, Connecticut, June 1991. To appear in the SIGPLAN Notices.
22. C. K. Holst and J. Hughes. Towards binding time improvement for free. In Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors, *Functional Programming, Glasgow 1990. Workshops in Computing*, pages 83–100. Springer-Verlag, August 1990.
23. J. Hughes and J. Launchbury. Towards relating forward and backwards analyses. In Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors, *Functional Programming, Glasgow 1990. Workshops in Computing*, pages 101–113. Springer-Verlag, August 1990.
24. T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In J.-P. Jouannaud, editor, *Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 190–203. Springer-Verlag, 1985.
25. N. D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France*, volume 202 of *Lecture Notes in Computer Science*, pages 124–140. Springer-Verlag, 1985.
26. N. D. Jones, P. Sestoft, and H. Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.
27. J. Jørgensen. Generating a pattern matching compiler by partial evaluation. In Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors, *Functional Programming, Glasgow 1990. Workshops in Computing*, pages 177–195. Springer-Verlag, August 1990.
28. U. Jørring and W. L. Scherlis. Compilers and staging transformations. In *ACM Symposium on Principles of Programming Languages*, pages 86–96, 1986.
29. J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.
30. 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.
31. J. Launchbury. *Projection Factorisation in Partial Evaluation*. PhD thesis, Department of Computing Science, University of Glasgow, Scotland, January 1990.
32. T. Mogensen. Partially static structures in a self-applicable partial evaluator. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–348. North-Holland, 1988.

33. T. Mogensen. *Binding Time Aspects of Partial Evaluation*. PhD thesis, DIKU, University of Copenhagen, Denmark, March 1989.
34. T. Mogensen. Separating binding times in language specifications. In *FPCA '89, 4th International Conference on Functional Programming Languages and Computer Architecture*, pages 12–25. ACM Press, 1989.
35. P. Mosses. *Action Semantics*. Cambridge University Press, 1991. draft textbook, in preparation.
36. F. Nielson. A denotational framework for data flow analysis. *Acta Informatica*, 18:265–287, 1982.
37. H. R. Nielson and F. Nielson. Eureka definitions for free! or disagreement points for fold/unfold transformations. In N. D. Jones, editor, *ESOP'90, 3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 291–305. Springer-Verlag, 1990.
38. G. D. Plotkin. Call-by-name, call-by-value and the λ -calculus. *Theoretical Computer Science*, 1:125–159, 1975.
39. J. Rees and W. Clinger, editors. Revised³ report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37–79, December 1986.
40. J. Reynolds. Definitional interpreters for higher order programming languages. In *ACM National Conference*, pages 717–740, 1972.
41. 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.
42. O. Shivers. The semantics of Scheme control-flow analysis. In *Proceedings of the first ACM SIGPLAN and IFIP Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, New Haven, Connecticut, June 1991. To appear in the SIGPLAN Notices.
43. D. A. Smith. Partial evaluation of pattern matching in CLP domains. In *Proceedings of the first ACM SIGPLAN and IFIP Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, New Haven, Connecticut, June 1991. To appear in the SIGPLAN Notices.
44. G. L. Steele, Jr. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.
45. V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
46. P. Wadler. Deforestation: Transforming programs to eliminate trees. In H. Ganzinger, editor, *ESOP'88, 2nd European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*. Springer-Verlag, 1988.
47. P. Wadler. Theorems for free! In *FPCA '89, 4th International Conference on Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, 1989.
48. M. Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, January 1980.
49. M. Wand. From interpreter to compiler: A representational derivation. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*, pages 306–324, 1985.
50. D. Weise and E. Ruf. Computing types during program specialization. Technical Report 441, Stanford University, Stanford, USA, August 1990.