Math. Struct. in Comp. Science (1996), *vol.* 00, *pp.* ??–?? Copyright © Cambridge University Press

A Computational Formalization for Partial Evaluation

JOHN HATCLIFF[1] and OLIVIER DANVY[2]

[1] *Department of Computer Science, Oklahoma State University*
219 Mathematical Sciences, Stillwater, OK, 74078-1053, USA
E-mail: hatcliff@a.cs.okstate.edu
[2] **BRICS**†
Department of Computer Science, Aarhus University
Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark
E-mail: danvy@brics.dk

Received December 1995, revised August 1996

We formalize a partial evaluator for Eugenio Moggi's computational metalanguage. This formalization gives an evaluation-order independent view of binding-time analysis and program specialization, including a proper treatment of call unfolding, and enables us to express the essence of "control-based binding-time improvements" for let expressions. Specifically, we prove that the binding-time improvements given by "continuation-based specialization" can be expressed in the metalanguage via monadic laws.

1. Introduction

Partial evaluation is a program-transformation technique for specializing programs, based on propagating constant values and folding constant expressions (Consel and Danvy 1993; Jones et al. 1993). Over the last ten years, it has been organized as a two-phase process: binding-time analysis and program specialization. Binding-time analysis classifies which parts of the source program can be computed statically, i.e., at partial-evaluation time. Program specialization carries out these static computations and residualizes the other computational steps. This organization in two phases thus makes it clear that the more parts of a source program are static, the more this source program is specialized. Over the last five years, various pre-transformations have been investigated that "improve binding times" (making binding-time analysis classify more parts are static) and thus increase specialization.

Our goal is to formalize both the two-phase process of partial evaluation and binding-time improvements, using Moggi's computational metalanguage.

Moggi's computational metalanguage distinguishes *values* (terms with no remaining computation steps) and *computations* (terms with remaining computation steps). This makes it possible to express a variety of evaluation strategies (including call-by-name

† Basic Research in Computer Science,
Centre of the Danish National Research Foundation.

and call-by-value). In addition, programs can be parameterized with various notions of computations, expressed as computational monads. Monads are often claimed to make reasoning about programs easier — but very few applications actually exploit the monadic laws.

In this paper, we illustrate that the computational meta-language can be a useful framework for partial evaluation. It allows a clear distinction between static computation steps (to be performed at specialization time) and dynamic computation steps (to be residualized at specialization time and performed at run time). Moreover, it allows an evaluation-order independent view of binding-time analysis and program specialization.

To this end, we present a PCF-like version of the computational metalanguage, and give it a structural operational semantics. We specify binding-time analysis as a non-standard type inference, and then a specializer using structural operational semantics. We prove the correctness of the binding-time analysis and of the specializer.

The computational metalanguage enables us to formalize existing techniques in partial evaluation — namely the linguistic device of let insertion, and the partial-evaluation techniques of control-based binding-time improvement and continuation-based specialization, which we achieve by incorporating the monadic laws into our binding-time analysis and our specializer. We prove the equivalence between a continuation-based partial evaluator and our partial evaluator that applies the monadic laws. This formalizes binding-time improvements independently of any particular evaluation order.

We believe that the computational metalanguage also enables new insights and techniques, e.g., to process computational effects. Let us first review each of these points, before outlining the rest of the paper.

1.1. *Call unfolding and computation duplication*

A partial evaluator unfolds function calls. Call unfolding is thus necessary but in general it is unsound under call-by value. It is necessary to expose opportunities for constant propagation and folding. It is unsound under call-by-value because it may alter termination properties and duplicate computations. Since Similix (Bondorf and Danvy 1991), virtually all call-by-value partial evaluators insert a let expression for every dynamic (i.e., unknown) parameter, at each unfolding point. This insertion ensures sound call unfolding: termination properties are preserved and both code and computation duplications are avoided.

Moggi's computational metalanguage specifies the order of computations through let expressions. By expressing source programs in this metalanguage, let expressions appear naturally. By residualizing these let expressions, sound call unfolding is achieved naturally.

For example, the call-by-value program

$$\lambda z\,.\,(\lambda(x,y)\,.\,(x+1)-(y+y))\,@\,(10, z\times z)$$

can be specialized as follows. The inner β-redex can be reduced (noting application with the infix operator "@"), and the left-most addition can be computed. The residual program reads:

$$\lambda z\,.\,11 - ((z \times z) + (z \times z))$$

It is unsatisfactory because the computation of $z \times z$ has been duplicated. As mentioned above (Bondorf and Danvy 1991), call-by-value partial evaluators insert residual let expressions to name dynamic expressions that should not be duplicated. With such a strategy, the residual program reads:

$$\lambda z\,.\,\text{let}\,a = z \times z\,\text{in}\,11 - (a + a)$$

This solution of inserting let expressions is effective, but ad hoc, in that there were no corresponding let expressions in the source program. In contrast, let expressions are an integral part of Moggi's computational metalanguage. Using this metalanguage, the source program can be rewritten as follows.

$$\begin{array}{l} \lambda z\,.\,\mathsf{let}\,a \Leftarrow z \times z \\ \quad \mathsf{in}\,(\lambda(x, y)\,.\,\mathsf{let}\,v_1 \Leftarrow x + 1 \\ \qquad\qquad\qquad \mathsf{in}\,\mathsf{let}\,v_2 \Leftarrow y + y \\ \qquad\qquad\qquad\quad \mathsf{in}\,v_1 - v_2)\,@\,(10, a) \end{array}$$

Specializing this term amounts to β-reducing function applications (which is *always* sound because their argument is a value, not a computation), δ-reducing static operations, and unfolding let expressions that bind a unit computation.

Presently, the outer let expression is not unfolded because z is dynamic (its value is unknown) and the function application is β-reduced. This leads to:

$$\begin{array}{l} \lambda z\,.\,\mathsf{let}\,a \Leftarrow z \times z \\ \quad \mathsf{in}\,\mathsf{let}\,v_1 \Leftarrow 10 + 1 \\ \qquad \mathsf{in}\,\mathsf{let}\,v_2 \Leftarrow a + a \\ \qquad\quad \mathsf{in}\,v_1 - v_2 \end{array}$$

The static addition is performed, which leads to:

$$\begin{array}{l} \lambda z\,.\,\mathsf{let}\,a \Leftarrow z \times z \\ \quad \mathsf{in}\,\mathsf{let}\,v_1 \Leftarrow \mathsf{unit}\,11 \\ \qquad \mathsf{in}\,\mathsf{let}\,v_2 \Leftarrow a + a \\ \qquad\quad \mathsf{in}\,v_1 - v_2 \end{array}$$

The let expression declaring v_1 is unfolded, and there is no other opportunity for static reduction. The result essentially coincides with the residual program above.

1.2. *Control-based binding-time improvements*

The structure of source programs influences the precision of binding-time analysis and thus the effectiveness of program specialization. A panoply of "binding-time improvements" aiming at restructuring source programs has been developed (Jones et al. 1993, Chapter 12) but they have not been formalized.

Essentially, a partial evaluator propagates static values into static contexts, where this propagation gives rise to a static computation. Binding-time analysis thus refines the usual notions of data flow and control flow into a static and a dynamic data flow, and a static and a dynamic control flow. Data-flow and control-flow binding-time improvements respectively amount to improve the static data flow and the static control flow of a source program. The data-flow aspects have been recently clarified (Danvy et al. 1995; Danvy et al. 1996), but the control-flow aspects still remain a research topic (Bondorf 1992; Bondorf and Dussart 1994; Consel and Danvy 1991; Holst and Gomard 1991; Lawall and Danvy 1994).

The program calculus for Moggi's computational metalanguage includes *monadic laws*. Let us show how a partial evaluator applying monadic laws to restructure source programs captures the essence of control-based binding-time improvements.

For example, the program

$$\lambda f\,.\,((\lambda x\,.\,2)\,@\,(f\,@\,1)) + 3$$

can be specialized as follows. The inner β-redex can be reduced but because its argument is dynamic, a let expression must be inserted. The result reads:

$$\lambda f\,.\,(\mathrm{let}\; x = f\,@\,1\;\mathrm{in}\;2) + 3$$

It would be unsound to unfold the let expression and discard $f\,@\,1$ from the residual program because f could have a computational effect (e.g., divergence). A partial evaluator needs to perform some contorsions to move the context of the let expression $[\cdot] + 3$ in its body, yielding

$$\lambda f\,.\,\mathrm{let}\; x = f\,@\,1\;\mathrm{in}\;5$$

Such restructurings are referred to as "control-based binding-time improvements" since they alter the control flow of the program so as to improve binding times. They are usually achieved by maintaining an explicit representation of control, using continuations. Specializers incorporating these techniques are known as "continuation-based specializers" (Bondorf 1992; Jones et al. 1993; Lawall and Danvy 1994). Essentially they mimic one-pass CPS transformations (Danvy and Filinski 1992).

We formalize this technique with the associativity of let expressions in Moggi's computational metalanguage. Encoding the term above in the metalanguage yields:

$$\begin{array}{l}\lambda f\,.\,\mathsf{let}\; v_1 \Leftarrow \mathsf{let}\; v_2 \Leftarrow f\,@\,1 \\ \qquad\qquad\qquad\quad \mathsf{in}\;(\lambda x\,.\,\mathsf{unit}\;2)\,@\,v_2 \\ \qquad \mathsf{in}\; v_1 + 3\end{array}$$

Static reduction of the β-redex yields:

$$\begin{array}{l}\lambda f\,.\,\mathsf{let}\; v_1 \Leftarrow \mathsf{let}\; v_2 \Leftarrow f\,@\,1 \\ \qquad\qquad\qquad\quad \mathsf{in}\;\mathsf{unit}\;2 \\ \qquad \mathsf{in}\; v_1 + 3\end{array}$$

Reassociating the let expression yields:

$$\begin{array}{l}\lambda f\,.\,\mathsf{let}\; v_2 \Leftarrow f\,@\,1 \\ \qquad \mathsf{in}\;\mathsf{let}\; v_1 \Leftarrow \mathsf{unit}\;2 \\ \qquad\quad \mathsf{in}\; v_1 + 3\end{array}$$

Unfolding the inner let and statically reducing the addition yields:

$$\begin{array}{l}\lambda f\,.\,\mathsf{let}\ v_2 \Leftarrow f\,@\,1\\ \quad\mathsf{in\ unit}\ 5\end{array}$$

The result essentially coincides with the residual program above.

1.3. *Computational effects*

Since Similix (Bondorf and Danvy 1991), partial evaluators classify computational effects (I/O, state, etc.) as dynamic computations. The specializer's only duty is to maintain their order and to ensure that none disappears or is duplicated.

The computational metalanguage's *raison d'être* is to provide a modular specification of computational effects. Using this medium, we originally planned to provide a sound treatment of side-effects that would be more effective than systematic residualization. We envisioned for example to split the store in regions: some side-effecting operations could then be classified as static and performed statically (e.g., a symbol table in an interpreter could be processed statically in a compiler) while others would be classified as dynamic and delayed until runtime (e.g., an error message associated with division by 0). This line of work is currently being pursued independently by Dussart and Thiemann (Dussart and Thiemann 1996).

1.4. *This paper*

The rest of this paper is organized as follows. Section 2 presents a PCF-like version of the computational metalanguage, Λ_{ml}. We give it a structural operational semantics. Section 3 presents an offline partial evaluator for Λ_{ml}. Section 4 addresses call unfolding. In Section 5, we return to the monadic laws and incorporate them into the specializer — adjusting the binding-time analysis accordingly. In Section 6, we prove the equivalence between a continuation-based partial evaluator and our partial evaluator that applies the monadic laws. After a review of related work in Section 7, Section 8 concludes.

2. The computational meta-language

2.1. *Syntax*

Figure 1 presents the language Λ_{ml} based on Moggi's *computational meta-language* (Moggi 1991).[†] The typing system of Λ_{ml} captures the distinction between *values* (terms with no remaining computation steps) and *computations* (terms with remaining computation steps). Types of the form nat and $\tau_1 \rightarrow \tau_2$ are *value types*. Accordingly, the rules for numerals and abstractions belong to the introduction rules for value types. Types of the form $\widetilde{\tau}$ are *computation types*. All functions have computational co-domains. Thus,

[†] We have added **fix**, **succ**, **pred**, and **if0** to Moggi's published description (Moggi 1991).

$$
\begin{array}{rcl}
e & \in & \mathit{Terms}[\Lambda_{ml}] \\
e & ::= & x \mid \ulcorner n \urcorner \mid \mathsf{succ}\, e \mid \mathsf{pred}\, e \mid \mathsf{if0}\, e_1\, e_2\, e_3 \mid \lambda x\,.\, e \mid e_0 \,@\, e_1 \mid \mathsf{fix}\, x.\, e \mid \\
 & & \mathsf{unit}\, e \mid \mathsf{let}\, x \Leftarrow e_1 \,\mathsf{in}\, e_2
\end{array}
$$

$$
\begin{array}{rcl}
\tau & \in & \mathit{Types}[\Lambda_{ml}] \\
\tau & ::= & \mathsf{nat} \mid \tau_1 \rightarrow \widetilde{\tau_2} \mid \widetilde{\tau}
\end{array}
\qquad\qquad
\begin{array}{rcl}
\Gamma & \in & \mathit{Assumptions}[\Lambda_{ml}] \\
\Gamma & ::= & \cdot \mid \Gamma, x : \tau
\end{array}
$$

$$
\Gamma \vdash_{ml} \ulcorner n \urcorner : \mathsf{nat}
\qquad
\frac{\Gamma \vdash_{ml} e : \mathsf{nat}}{\Gamma \vdash_{ml} \mathsf{succ}\, e : \widetilde{\mathsf{nat}}}
\qquad
\frac{\Gamma \vdash_{ml} e : \mathsf{nat}}{\Gamma \vdash_{ml} \mathsf{pred}\, e : \widetilde{\mathsf{nat}}}
$$

$$
\Gamma \vdash_{ml} x : \Gamma(x)
\qquad
\frac{\Gamma \vdash_{ml} e_1 : \mathsf{nat} \qquad \Gamma \vdash_{ml} e_2 : \widetilde{\tau} \qquad \Gamma \vdash_{ml} e_3 : \widetilde{\tau}}{\Gamma \vdash_{ml} \mathsf{if0}\, e_1\, e_2\, e_3 : \widetilde{\tau}}
$$

$$
\frac{\Gamma, x : \tau_1 \vdash_{ml} e : \widetilde{\tau_2}}{\Gamma \vdash_{ml} \lambda x\,.\, e : \tau_1 \rightarrow \widetilde{\tau_2}}
\qquad
\frac{\Gamma \vdash_{ml} e_0 : \tau_1 \rightarrow \widetilde{\tau_2} \qquad \Gamma \vdash_{ml} e_1 : \tau_1}{\Gamma \vdash_{ml} e_0 \,@\, e_1 : \widetilde{\tau_2}}
\qquad
\frac{\Gamma, x : \widetilde{\tau} \vdash_{ml} e : \widetilde{\tau}}{\Gamma \vdash_{ml} \mathsf{fix}\, x.\, e : \widetilde{\tau}}
$$

$$
\text{monadic constructs:}\quad
\frac{\Gamma \vdash_{ml} e : \tau}{\Gamma \vdash_{ml} \mathsf{unit}\, e : \widetilde{\tau}}
\qquad
\frac{\Gamma \vdash_{ml} e_1 : \widetilde{\tau_1} \qquad \Gamma, x : \tau_1 \vdash_{ml} e_2 : \widetilde{\tau_2}}{\Gamma \vdash_{ml} \mathsf{let}\, x \Leftarrow e_1 \,\mathsf{in}\, e_2 : \widetilde{\tau_2}}
$$

Fig. 1. The computational meta-language Λ_{ml}

$$
\begin{array}{rcl}
\mathsf{succ}\, \ulcorner n \urcorner & \longmapsto_i & \mathsf{unit}\, \ulcorner n+1 \urcorner \\
(\lambda x\,.\, e_0) \,@\, e_1 & \longmapsto_i & e_0[x := e_1] \\
\mathsf{fix}\, x.\, e & \longmapsto_i & e[x := \mathsf{fix}\, x.\, e] \\
\mathsf{let}\, x \Leftarrow \mathsf{unit}\, e_1 \,\mathsf{in}\, e_2 & \longmapsto_i & e_2[x := e_1]
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{pred}\, \ulcorner n+1 \urcorner & \longmapsto_i & \mathsf{unit}\, \ulcorner n \urcorner \\
\mathsf{pred}\, \ulcorner 0 \urcorner & \longmapsto_i & \mathsf{unit}\, \ulcorner 0 \urcorner \\
\mathsf{if0}\, \ulcorner 0 \urcorner\, e_2\, e_3 & \longmapsto_i & e_2 \\
\mathsf{if0}\, \ulcorner n+1 \urcorner\, e_2\, e_3 & \longmapsto_i & e_3
\end{array}
$$

$$
\frac{e \longmapsto_i e'}{E_i[e] \longmapsto_i E_i[e']}
\qquad\qquad
E_i \;::=\; \mathsf{let}\, x \Leftarrow [\cdot] \,\mathsf{in}\, e_2
$$

Fig. 2. Interpretation steps for Λ_{ml}

all applications have a computation type — capturing the fact that evaluating a function application always requires one or more computational steps.

The *monadic constructs* are used to make the computational process explicit.[‡] Intuitively, $\mathsf{unit}\, e$ is a trivial computation that simply yields the value of e. $\mathsf{let}\ x \Leftarrow e_1\ \mathsf{in}\ e_2$ forces the evaluation of e_1. If that evaluation terminates, the resulting value is substituted for x in e_2, and evaluation continues with the modified version of e_2.

We identify terms up to renaming of bound variables (i.e., up to α-equivalence) and use standard notation and conventions for substitution, free variables, contexts, etc. (Barendregt 1984). We write $e_1 \equiv e_2$ when e_1 and e_2 are α-equivalent.

We represent simultaneous substitutions $e[x_1 := e_1\,, \ldots, x_n := e_n]$ using a substitution function $\sigma : \mathit{Terms}[\Lambda_{ml}] \rightarrow \mathit{Terms}[\Lambda_{ml}]$ whose application is denoted $e\sigma$. A substitution σ is *closed* if, for all $x \in \mathrm{dom}\ \sigma$, $x\sigma$ is a closed term.

We write $\Gamma \vdash_{ml} e_1\,, e_2 : \tau$ when both $\Gamma \vdash_{ml} e_1 : \tau$ and $\Gamma \vdash_{ml} e_2 : \tau$. A closed substitution σ is *compatible* with an assumption Γ, if for all $x \in \mathit{dom}\ \Gamma$, $\cdot \vdash_{ml} x\sigma : \Gamma(x)$. *Programs* are closed terms with type $\widetilde{\mathsf{nat}}$.

2.2. *Operational semantics*

Figure 2 presents single-step evaluation rules for Λ_{ml}.[§] Axioms such as $\mathsf{succ}\, \ulcorner n \urcorner \longmapsto_i \mathsf{unit}\, \ulcorner n+1 \urcorner$ define basic computation steps. The single inference rule describes contexts in which evaluation steps may occur.

The following lemma gives evaluation properties for closed terms at each type τ. The intuition is that each well-typed term (a) is a canonical term of the corresponding type, or (b) can undergo an evaluation step that preserves typing. In particular, there are no "stuck" (Plotkin 1975, p. 151) terms.[¶]

Lemma 1. (interpretation properties)
— If $\cdot \vdash_{ml} e : \mathsf{nat}$ then $e \equiv \ulcorner n \urcorner$ for some number n.
— If $\cdot \vdash_{ml} e : \tau_1 \rightarrow \widetilde{\tau_2}$ then $e \equiv \lambda x\,.\, e'$ and $x : \tau_1 \vdash_{ml} e' : \widetilde{\tau_2}$.
— If $\cdot \vdash_{ml} e : \widetilde{\tau}$ then exactly one of the following statements holds:

1 $e = \mathsf{unit}\, e'$ and $\cdot \vdash_{ml} e' : \tau$.

2 $e \longmapsto_i e'$ and $\cdot \vdash_{ml} e' : \widetilde{\tau}$.

Proof. by induction over the height of the derivation of $\cdot \vdash_{ml} e : \tau$ relying on the property that if $\Gamma, x : \tau_1 \vdash_{ml} e_0 : \tau_0$ and $\Gamma \vdash_{ml} e_1 : \tau_1$ then $\Gamma \vdash_{ml} e_0[x := e_1] : \tau_0$. □

[‡] The exact connection to the structure of a monad can be found in Moggi's original work (Moggi 1991, page 61).

[§] Moggi originally gave a categorical interpretation for his computational meta-language (Moggi 1991). Crole and Pitts extended Moggi's work with a fix-point operator and an associated logic (Crole and Pitts 1992). Later, Gordon developed an elegant operational theory for the meta-language with a fixpoint operator and inductive and coinductive types (Gordon 1993). Λ_{ml} is basically a sub-language of Gordon's — except that we include directly type **nat** and associated operations whereas Gordon constructs them via inductive types. Here, we follow Gordon and our previous work (Hatcliff 1994; Hatcliff and Danvy 1994) and present a structural operational semantics for Λ_{ml}.

[¶] For example, the untypable term $\ulcorner 3 \urcorner @ \ulcorner 2 \urcorner$ is stuck: it cannot undergo an evaluation step and it is not a proper canonical term for any type.

It is easy to check that $e \longmapsto_i e'$ and $e \longmapsto_i e''$ implies $e' \equiv e''$. This justifies the definition of the following (partial) function in terms of the reflexive transitive closure of $\longmapsto_i$.

Definition 1. (interpreter) For $\cdot \vdash_{ml} e : \widetilde{\mathsf{nat}}$,

$$\mathit{int}\, e \;=\; \ulcorner n \urcorner \quad \mathit{iff} \quad e \longmapsto_i^* \mathsf{unit}\, \ulcorner n \urcorner$$

We write $\mathit{int}\, e\!\downarrow$ when $\mathit{int}\ e$ is defined and $\mathit{int}\, e\!\uparrow$ when $\mathit{int}\ e$ is undefined. As a consequence of Lemma 1, $\mathit{int}\, e\!\uparrow$ implies that e heads an infinite sequence of computation steps.

Observing termination of terms in program contexts gives the following notion of operational approximation, which in turn, induces a notion of operational equivalence.[||]

Definition 2. (operational approximation) For $\Gamma \vdash_{ml} e_1\,,\, e_2 : \tau$, and for all contexts C such that $C[e_1]$ and $C[e_2]$ are programs,

$$e_1 \preceq e_2 \quad \mathit{iff} \quad \mathit{int}\, C[e_1]\!\downarrow \text{ implies } \mathit{int}\, C[e_2]\!\downarrow$$

Definition 3. (operational equivalence) For $\Gamma \vdash_{ml} e_1\,,\, e_2 : \tau$,

$$e_1 \approx e_2 \quad \mathit{iff} \quad e_1 \preceq e_2 \text{ and } e_2 \preceq e_1$$

Note that if $\cdot \vdash_{ml} e_1\,,\, e_2 : \widetilde{\mathsf{nat}}$ and $e_1 \approx e_2$, then $\mathit{int}\, e_1$ and $\mathit{int}\, e_2$ are both undefined, or else both are defined and there exists a number n such that $\mathit{int}\, e_1 \equiv \ulcorner n \urcorner \equiv \mathit{int}\, e_2$.

2.3. *Equational reasoning*

Figure 3 presents notions of reduction for the computational meta-language Λ_{ml}. $\longrightarrow$ also denotes construct-compatible one-step reduction, $\longrightarrow\!\!\!\!\!\rightarrow$ denotes the reflexive, transitive closure of $\longrightarrow$, and $=_{ml}$ denotes the smallest equivalence relation generated by $\longrightarrow$ (Barendregt 1984).

Theorem 1. (soundness of calculus) For $\Gamma \vdash_{ml} e_1\,,\, e_2 : \tau$,

$$e_1 =_{ml} e_2 \;\Rightarrow\; e_1 \approx e_2$$

Proof. The proof follows by a straightforward adaptation of Gordon's operational theory (Gordon 1993, Chapters 3,4,5). □

2.4. *Encoding evaluation strategies in Λ_{ml}*

A variety of evaluation strategies for a PCF-like language Λ can be encoded in Λ_{ml} due to Λ_{ml}'s explicit distinction between values and computations. In previous work (Hatcliff 1994; Hatcliff and Danvy 1994), we proved the correctness of an encoding $\mathcal{E}_n$ of a call-by-name version of Λ into Λ_{ml}, and an encoding $\mathcal{E}_v$ of a call-by-value version of Λ into Λ_{ml}. These encodings can be found in Appendix A.2.[**]

[||] It is sufficient to observe termination since one can always distinguish between numerals using conditional expressions.

[**] These encodings are based on similar encodings given by e.g., Moggi (Moggi 1991) and Wadler (Wadler 1992b).

$$
\begin{array}{rcl@{\qquad}rcl}
\mathsf{succ}\,\ulcorner n\urcorner & \longrightarrow & \mathsf{unit}\,\ulcorner n+1\urcorner & \mathsf{pred}\,\ulcorner n+1\urcorner & \longrightarrow & \mathsf{unit}\,\ulcorner n\urcorner \\
(\lambda x\,.\,e_0)\,@\,e_1 & \longrightarrow & e_0[x := e_1] & \mathsf{pred}\,\ulcorner 0\urcorner & \longrightarrow & \mathsf{unit}\,\ulcorner 0\urcorner \\
\mathsf{fix}\,x.\,e & \longrightarrow & e[x := \mathsf{fix}\,x.\,e] & \mathsf{if0}\,\ulcorner 0\urcorner\,e_2\,e_3 & \longrightarrow & e_2 \\
 & & & \mathsf{if0}\,\ulcorner n+1\urcorner\,e_2\,e_3 & \longrightarrow & e_3
\end{array}
$$

Monadic reductions:

$$
\begin{array}{lrcll}
(\mathit{let}.\beta) & \mathsf{let}\ x \Leftarrow \mathsf{unit}\ e_1\ \mathsf{in}\ e_2 & \longrightarrow & e_2[x := e_1] & \\
(\mathit{let}.\eta) & \mathsf{let}\ x \Leftarrow e\ \mathsf{in}\ \mathsf{unit}\ x & \longrightarrow & e & \\
(\mathit{let.assoc}) & \mathsf{let}\ x_2 \Leftarrow \mathsf{let}\ x_1 \Leftarrow e_1 & \longrightarrow & \mathsf{let}\ x_1 \Leftarrow e_1 & x_1 \notin FV(e_3) \\
 & \mathsf{in}\ e_2 & & \mathsf{in}\ \mathsf{let}\ x_2 \Leftarrow e_2 & \\
 & \mathsf{in}\ e_3 & & \quad\mathsf{in}\ e_3 &
\end{array}
$$

Fig. 3. Notions of reduction *ml* for Λ_{ml}

In summary, call-by-name Λ functions are encoded as functions from computations to computations; call-by-value Λ functions are encoded as functions from values to computations.[††] Applications are translated as follows:

$$
\begin{array}{lrcl}
\textit{Call-by-name:} & \mathcal{E}_n\langle\!\![e_0\,@\,e_1]\!\!\rangle & = & \mathsf{let}\ x_0 \Leftarrow \mathcal{E}_n\langle\!\![e_0]\!\!\rangle\ \mathsf{in}\ x_0\,@\,\mathcal{E}_n\langle\!\![e_1]\!\!\rangle \\
\textit{Call-by-value:} & \mathcal{E}_v\langle\!\![e_0\,@\,e_1]\!\!\rangle & = & \mathsf{let}\ x_0 \Leftarrow \mathcal{E}_v\langle\!\![e_0]\!\!\rangle\ \mathsf{in}\ \mathsf{let}\ x_1 \Leftarrow \mathcal{E}_v\langle\!\![e_1]\!\!\rangle\ \mathsf{in}\ x_0\,@\,x_1
\end{array}
$$

Thus, evaluation of argument expressions in not forced in the call-by-name encoding, but is forced in the call-by-value encoding.

In general (i.e., in all encodings), a let is inserted around each computation step — making the computational structure of a program explicit. This property is crucial to the evaluation-order independent treatment of binding-time analysis and program specialization presented in the following section.

3. An offline partial evaluator for Λ_{ml}

A partial evaluator takes a *source program* p and a subset s of p's input, and produces a *residual program* p_s which is specialized with respect to s. The correctness of the partial evaluator implies that running p_s on p's remaining input d gives the same result as running p on the complete input s and d. The data s and d are often referred to as *static* and *dynamic* data (respectively) since s is fixed at specialization time whereas one may supply various data d during runs of p_s.

[††] Alternative call-by-value encodings exist that are similar to the encoding of call-by-value procedures in the call-by-name programming language Algol 60 (Hatcliff and Danvy 1994, Section 4).

The specialized program p_s is obtained from p by evaluating constructs that depend only on s, while rebuilding constructs that may depend on dynamic data. "Offline" partial evaluation accomplishes this in two phases: (1) a *binding-time analysis* phase, and (2) a *specialization* phase.

1 **Binding-time analysis**: Given assumptions about which program inputs are static and dynamic, binding-time analysis constructs an annotated program where each program construct is annotated with a *specialization directive* and a *specialization type*.

— **Specialization directives**: A construct is assigned a directive of *eliminable* if it depends only on static data and thus can be completely evaluated during the specialization phase. A construct is assigned a directive of *residual* if it may depend on dynamic data and thus must be reconstructed in the specialization phase.

— **Specialization types**: The specialization types (*a.k.a. binding times*) are the carriers of information during the analysis phase. They describe the "knownness" or the "unknownness" of expressions. This information is used to determine the specialization directives assigned to constructs. For example, if the argument of a destructor construct has a specialization type indicating that it is unknown, then that construct cannot be evaluated at specialization time and must be given a *residual* directive.

2 **Specialization**: During the specialization phase, the specializer simply follows the directives assigned during binding-time analysis: eliminable constructs are evaluated (and thus eliminated); residual constructs are reconstructed (and thus appear in the residual program).[†]

3.1. *Binding-time analysis*

We first outline how binding-time information is expressed in a program annotated with specialization directives and types. Following this, we give a binding-time logic that determines which annotations are appropriate for a given source language term.

3.1.1. *Specialization directives* A binding-time analysis for Λ_{ml} associates each Λ_{ml} term with a term in the annotated language Λ_{ml}^{bt} of Figure 4. Eliminable terms are non-underlined; residual terms are underlined. Identifiers are not annotated since the appropriate information can be determined from the environment. A coercion construct lift is added to Λ_{ml}^{bt} to residualize the result of evaluating an eliminable term. This allows static computation to occur in a residual context.[‡] A term $w \in \Lambda_{ml}^{bt}$ is *completely residual* if it consists of only underlined constructs and identifiers. *Residual-terms*$[\Lambda_{ml}^{bt}]$ denotes

[†] Following other formal treatments of partial evaluation (Jones et al. 1993; Palsberg 1993; Wand 1993), we simplify the presentation by omitting folding and generalization strategies.

[‡] We restrict lifting to base types for simplicity, as we wish to formalize control-flow binding-time improvements. Defining lift at higher types would enable data-flow binding-time improvements, as investigated elsewhere (Danvy et al. 1995; Danvy et al. 1996). This definition is also interesting on its own (Danvy 1996).

$$
\begin{array}{rcl}
w & \in & \mathit{Terms}[\Lambda_{ml}^{bt}] \\
w & ::= & x \mid \\
 & & \ulcorner n \urcorner \mid \mathsf{succ}\, w \mid \mathsf{pred}\, w \mid \mathsf{if0}\, w_1\, w_2\, w_3 \mid \lambda x\,.\,w \mid w_0 \mathbin{@} w_1 \mid \mathsf{fix}\, x.\,w \mid \\
 & & \mathsf{unit}\, w \mid \mathsf{let}\, x \Leftarrow w_1\, \mathsf{in}\, w_2 \mid \\
 & & \underline{n} \mid \underline{\mathsf{succ}}\, w \mid \underline{\mathsf{pred}}\, w \mid \underline{\mathsf{if0}}\, w_1\, w_2\, w_3 \mid \underline{\lambda} x\,.\,w \mid w_0 \mathbin{\underline{@}} w_1 \mid \underline{\mathsf{fix}}\, x.\,w \mid \\
 & & \underline{\mathsf{unit}}\, w \mid \underline{\mathsf{let}}\, x \Leftarrow w_1\, \underline{\mathsf{in}}\, w_2 \mid \mathsf{lift}\, w
\end{array}
$$

Fig. 4. The binding-time annotated meta-language Λ_{ml}^{bt}

$$
\begin{array}{rclcrclcrcl}
\varphi_{\mathsf{nat}} & \in & \mathit{STypes}[\mathsf{nat}] & & \varphi_{\tau_1 \to \widetilde{\tau_2}} & \in & \mathit{STypes}[\tau_1 \to \widetilde{\tau_2}] & & \varphi_{\widetilde{\tau}} & \in & \mathit{STypes}[\widetilde{\tau}] \\
\varphi_{\mathsf{nat}} & ::= & \mathsf{s} \mid \mathsf{d} & & \varphi_{\tau_1 \to \widetilde{\tau_2}} & ::= & \varphi_{\tau_1} \to \varphi_{\widetilde{\tau_2}} \mid \mathsf{d} & & \varphi_{\widetilde{\tau}} & ::= & \widetilde{\varphi_\tau} \mid \mathsf{d}
\end{array}
$$

$$
\begin{array}{rclcrcl}
\tau & \in & \mathit{Types}[\Lambda_{ml}^{bt}] & & \Gamma & \in & \mathit{Assumptions}[\Lambda_{ml}^{bt}] \\
\tau & ::= & \mathsf{nat} \mid \tau_1 \to \widetilde{\tau_2} \mid \widetilde{\tau} & & \Gamma & ::= & \cdot \mid \Gamma, x : \tau[\varphi_\tau]
\end{array}
$$

Fig. 5. Specialization types and binding-time assumptions for Λ_{ml}^{bt}

the set of completely residual terms. Intuitively, the specializer will output completely residual terms — all eliminable constructs will have been evaluated (this will be proved in Section 3.2).

3.1.2. *Specialization types* Figure 5 presents a τ-indexed family of specialization types (*STypes*) for Λ_{ml}^{bt} (we omit type indices on specialization types when they can be inferred from the context). A specialization type φ is *dynamic* if $\varphi = \mathsf{d}$; otherwise φ is *static*.

- $\mathsf{s} \in \mathit{STypes}[\mathsf{nat}]$ will tag expressions of type nat that are guaranteed to evaluate to known data (i.e., numerals).
- $\mathsf{d} \in \mathit{STypes}[\mathsf{nat}]$ will tag expressions of type nat whose evaluation may depend on unknown data and thus cannot be guaranteed to evaluate to numerals. However, because nat is a value type, one does know that the tagged expression will denote a value when dynamic data is supplied.
- $\varphi_{\tau_1} \to \varphi_{\widetilde{\tau_2}} \in \mathit{STypes}[\tau_1 \to \widetilde{\tau_2}]$ will tag expressions of type $\tau_1 \to \widetilde{\tau_2}$ that are guaranteed to evaluate to known data (i.e., abstractions).
- $\mathsf{d} \in \mathit{STypes}[\tau_1 \to \widetilde{\tau_2}]$ will tag expressions of type $\tau_1 \to \widetilde{\tau_2}$ whose evaluation may depend on unknown data and thus cannot be guaranteed to evaluate to an abstraction. However, because $\tau_1 \to \widetilde{\tau_2}$ is a value type, one does know that the tagged expression will denote a value when dynamic data is supplied.
- $\widetilde{\varphi} \in \mathit{STypes}[\widetilde{\tau}]$ will tag expressions of type $\widetilde{\tau}$ that are guaranteed to either diverge or to evaluate to a trivial computation (i.e., $\mathsf{unit}\, e$ for some $e \in \Lambda_{ml}$ of type τ).

— $\mathsf{d} \in \mathit{STypes}[\widetilde{\tau}]$ will tag expressions of type $\widetilde{\tau}$ whose evaluation may depend on unknown data and thus cannot be guaranteed to diverge or evaluate to a trivial computation.

Figure 5 also presents type assumptions for Λ^{bt}_{ml} variables. If $\Gamma, x:\tau[\mathsf{d}]$ then x is a *dynamic variable*; if $\Gamma, x:\tau[\varphi]$ where φ is static then x is a *static variable*. $\Gamma(x).\mathit{type}$ and $\Gamma(x).\mathit{spec\text{-}type}$ project types and specialization types from an assumption for $x \in \mathit{dom}\ \Gamma$.

3.1.3. *Binding-time analysis specification* Figures 6 and 7 present rules for deriving judgments of the form $\Gamma \vdash_{bt} e : \tau[w : \varphi_\tau]$. Derivable judgements specify constraints that an actual binding-time analysis algorithm must satisfy. Intuitively, if $\Gamma \vdash_{bt} e : \tau[w : \varphi_\tau]$, then given initial binding-time assumptions Γ that indicate which free variables are static or dynamic, a binding-time analysis algorithm maps $e \in \mathit{Terms}[\Lambda_{ml}]$ of type $\tau \in \mathit{Types}[\Lambda_{ml}]$ to a directive annotated term $w \in \mathit{Terms}[\Lambda^{bt}_{ml}]$ of specialization type $\varphi_\tau \in \mathit{STypes}[\tau]$. Specifying the analysis in this way allows one to reason about correctness of the analysis independently of the actual algorithm — proving correctness with respect to the constraints implies that any algorithm satisfying the constraints will be correct.

$\lfloor\cdot\rfloor : \mathit{Terms}[\Lambda^{bt}_{ml}] \rightarrow \mathit{Terms}[\Lambda_{ml}]$ is an erasing function that removes annotations and lift constructs. $\lfloor\cdot\rfloor : \mathit{Assumptions}[\Lambda^{bt}_{ml}] \rightarrow \mathit{Assumptions}[\Lambda_{ml}]$ simply drops specialization types appearing in assumptions. $\lfloor\sigma\rfloor$ represents the component-wise application of $\lfloor\cdot\rfloor : \mathit{Terms}[\Lambda^{bt}_{ml}] \rightarrow \mathit{Terms}[\Lambda_{ml}]$. A closed Λ^{bt}_{ml} substitution σ is *compatible* with an assumption $\Gamma \in \mathit{Assumptions}[\Lambda^{bt}_{ml}]$, if for all $x \in \mathit{dom}\ \Gamma$, $\cdot \vdash_{ml} \lfloor x\sigma\rfloor : \Gamma(x).\mathit{type}[x\sigma : \Gamma(x).\mathit{spec\text{-}type}]$. For expressions $e \in \mathit{Terms}[\Lambda_{ml}]$, Γ_{bt} is a *binding-time assumption for* $\Gamma \vdash_{ml} e : \tau$, if $\lfloor\Gamma_{bt}\rfloor = \Gamma$. $\Gamma_{bt} = \Gamma_s \cup \Gamma_d$ represents the splitting of a binding-time assumption Γ_{bt} into its (necessarily disjoint) static and dynamic variable assumptions.

Definition 4. (binding-time analysis) A *binding-time analysis* is a function

$$\mathit{bta} : \mathit{Assumptions}[\Lambda^{bt}_{ml}] \rightarrow \mathit{Terms}[\Lambda_{ml}] \rightarrow \mathit{Terms}[\Lambda^{bt}_{ml}]$$

such that when Γ_{bt} is a binding-time assumption for $\Gamma \vdash_{ml} e : \widetilde{\mathsf{nat}}$,

$$\Gamma_{bt} \vdash_{bt} e : \widetilde{\mathsf{nat}}[(\mathit{bta}\,\Gamma_{bt}\ e) : \mathsf{d}].$$

The binding-time analysis rules specify a one-to-many relation between Λ_{ml} terms and Λ^{bt}_{ml} terms. Thus, there are many valid binding-time analysis functions — each varying in pratical effectiveness. One usually desires an analysis that gives as many eliminable terms as possible. However, it may be useful to deliberately annotate some eliminable terms as residual to ensure that specialization terminates (Jones et al. 1993). Our specification abstracts away from these orthogonal implementation issues.

In the other direction, an induction over the derivation of $\Gamma \vdash_{ml} e : \tau[w : \varphi]$ shows that $e \equiv \lfloor w\rfloor$. Thus, every well-annotated Λ^{bt}_{ml} term has exactly one Λ_{ml} term related to it — the Λ_{ml} term is the erasure of the Λ^{bt}_{ml} term.

3.2. *Specialization*

To specialize a binding-time analyzed program term w such that $\Gamma_{bt} = \Gamma_s \cup \Gamma_d \vdash_{bt} e : \widetilde{\mathsf{nat}}[w : \mathsf{d}]$, one supplies known data into the static variables via a substitution σ_s

$$\Gamma \vdash_{bt} x : \Gamma(x).\mathit{type}[x : \Gamma(x).\mathit{spec\text{-}type}]$$

$$\Gamma \vdash_{bt} \ulcorner n \urcorner : \mathsf{nat}[\ulcorner n \urcorner : \mathsf{s}] \qquad \Gamma \vdash_{bt} \ulcorner n \urcorner : \mathsf{nat}[\underline{\ulcorner n \urcorner} : \mathsf{d}]$$

$$\frac{\Gamma \vdash_{bt} e : \mathsf{nat}[w : \mathsf{s}]}{\Gamma \vdash_{bt} \mathsf{succ}\, e : \widetilde{\mathsf{nat}} \quad [\mathsf{succ}\, w : \widetilde{\mathsf{s}}]} \qquad \frac{\Gamma \vdash_{bt} e : \mathsf{nat}[w : \mathsf{d}]}{\Gamma \vdash_{bt} \mathsf{succ}\, e : \widetilde{\mathsf{nat}} \quad [\underline{\mathsf{succ}}\, w : \mathsf{d}]}$$

$$\frac{\Gamma \vdash_{bt} e : \mathsf{nat}[w : \mathsf{s}]}{\Gamma \vdash_{bt} \mathsf{pred}\, e : \widetilde{\mathsf{nat}} \quad [\mathsf{pred}\, w : \widetilde{\mathsf{s}}]} \qquad \frac{\Gamma \vdash_{bt} e : \mathsf{nat}[w : \mathsf{d}]}{\Gamma \vdash_{bt} \mathsf{pred}\, e : \widetilde{\mathsf{nat}} \quad [\underline{\mathsf{pred}}\, w : \mathsf{d}]}$$

$$\frac{\Gamma \vdash_{bt} e_1 : \mathsf{nat}[w_1 : \mathsf{s}] \qquad \Gamma \vdash_{bt} e_2 : \widetilde{\tau}[w_2 : \varphi] \qquad \Gamma \vdash_{bt} e_3 : \widetilde{\tau}[w_3 : \varphi]}{\Gamma \vdash_{bt} \mathsf{if0}\, e_1\, e_2\, e_3 : \widetilde{\tau}[\mathsf{if0}\, w_1\, w_2\, w_3 : \varphi]}$$

$$\frac{\Gamma \vdash_{bt} e_1 : \mathsf{nat}[w_1 : \mathsf{d}] \qquad \Gamma \vdash_{bt} e_2 : \widetilde{\tau}[w_2 : \mathsf{d}] \qquad \Gamma \vdash_{bt} e_3 : \widetilde{\tau}[w_3 : \mathsf{d}]}{\Gamma \vdash_{bt} \mathsf{if0}\, e_1\, e_2\, e_3 : \widetilde{\tau}[\underline{\mathsf{if0}}\, w_1\, w_2\, w_3 : \mathsf{d}]}$$

$$\frac{\Gamma, x : \tau_1[\varphi_1] \vdash_{bt} e : \widetilde{\tau_2}[w : \varphi_2]}{\Gamma \vdash_{bt} \lambda x \,.\, e : \tau_1 \rightarrow \widetilde{\tau_2}[\lambda x \,.\, w : \varphi_1 \rightarrow \varphi_2]}$$

$$\frac{\Gamma, x : \tau_1[\mathsf{d}] \vdash_{bt} e : \widetilde{\tau_2}[w : \mathsf{d}]}{\Gamma \vdash_{bt} \lambda x \,.\, e : \tau_1 \rightarrow \widetilde{\tau_2}[\underline{\lambda} x \,.\, w : \mathsf{d}]}$$

$$\frac{\Gamma \vdash_{bt} e_0 : \tau_1 \rightarrow \widetilde{\tau_2}[w_0 : \varphi_1 \rightarrow \varphi_2] \qquad \Gamma \vdash_{bt} e_1 : \tau_1[w_1 : \varphi_1]}{\Gamma \vdash_{bt} e_0 \,@\, e_1 : \widetilde{\tau_2}[w_0 \,@\, w_1 : \varphi_2]}$$

$$\frac{\Gamma \vdash_{bt} e_0 : \tau_1 \rightarrow \widetilde{\tau_2}[w_0 : \mathsf{d}] \qquad \Gamma \vdash_{bt} e_1 : \tau_1[w_1 : \mathsf{d}]}{\Gamma \vdash_{bt} e_0 \,@\, e_1 : \widetilde{\tau_2}[w_0 \,\underline{@}\, w_1 : \mathsf{d}]}$$

$$\frac{\Gamma, x : \widetilde{\tau}[\widetilde{\varphi}] \vdash_{bt} e : \widetilde{\tau}[w : \widetilde{\varphi}]}{\Gamma \vdash_{bt} \mathsf{fix}\, x.\, e : \widetilde{\tau}[\mathsf{fix}\, x.\, w : \widetilde{\varphi}]} \qquad \frac{\Gamma, x : \widetilde{\tau}[\mathsf{d}] \vdash_{bt} e : \widetilde{\tau}[w : \mathsf{d}]}{\Gamma \vdash_{bt} \mathsf{fix}\, x.\, e : \widetilde{\tau}[\underline{\mathsf{fix}}\, x.\, w : \mathsf{d}]}$$

Fig. 6. Binding-time constraints (part 1)

$$\frac{\Gamma \vdash_{bt} e : \tau[w : \varphi]}{\Gamma \vdash_{bt} \mathsf{unit}\, e : \widetilde{\tau}[\mathsf{unit}\, w : \widetilde{\varphi}]} \qquad \frac{\Gamma \vdash_{bt} e : \tau[w : \mathsf{d}]}{\Gamma \vdash_{bt} \mathsf{unit}\, e : \widetilde{\tau}[\underline{\mathsf{unit}}\, w : \mathsf{d}]}$$

$$\frac{\Gamma \vdash_{bt} e_1 : \widetilde{\tau_1}[w_1 : \widetilde{\varphi_1}] \quad \Gamma, x : \tau_1[\varphi_1] \vdash_{bt} e_2 : \widetilde{\tau_2}[w_2 : \varphi_2]}{\Gamma \vdash_{bt} \mathsf{let}\, x \Leftarrow e_1\, \mathsf{in}\, e_2 : \widetilde{\tau_2}[\mathsf{let}\, x \Leftarrow w_1\, \mathsf{in}\, w_2 : \varphi_2]}$$

$$\frac{\Gamma \vdash_{bt} e_1 : \widetilde{\tau_1}[w_1 : \mathsf{d}] \quad \Gamma, x : \tau_1[\mathsf{d}] \vdash_{bt} e_2 : \widetilde{\tau_2}[w_2 : \mathsf{d}]}{\Gamma \vdash_{bt} \mathsf{let}\, x \Leftarrow e_1\, \mathsf{in}\, e_2 : \widetilde{\tau_2}[\underline{\mathsf{let}}\, x \Leftarrow w_1\, \underline{\mathsf{in}}\, w_2 : \mathsf{d}]}(*)$$

$$\frac{\Gamma \vdash_{bt} e : \mathsf{nat}[w : \mathsf{s}]}{\Gamma \vdash_{bt} e : \mathsf{nat}[\mathsf{lift}\, w : \mathsf{d}]}$$

Fig. 7. Binding-time constraints (part 2)

compatible with Γ_s. The resulting term $w\sigma_s$ is then specialized using the rules of Figure 8 (which define the single-step specialization function $\longmapsto_s$).

— The interpretation rules allow interpreter steps to reduce eliminable (i.e., non-underlined) terms. This corresponds to the idea that partial evaluators are often described as having an interpreter component.

— The axiom lift coerces a static numeral to a dynamic numeral. This allows static values of type nat to occur in dynamic contexts.

— The compilation rules direct the activities of the specializer in dynamic contexts E_s. This corresponds to the fact that the non-interpretive component of the specializer simply constructs terms that appear in the residual program. In contexts E_s, r ranges over completely residual terms (i.e., terms containing only underlined constructs and free dynamic variables).

The following lemma gives specialization properties for the Λ_{ml}^{bt} terms satisfying judgements of the form $\Gamma_d \vdash_{bt} e : \tau[w : \varphi_\tau]$. The intuition is that each well-annotated term w (a) is a canonical annotated term of the appropriate type, or (b) is completely residual, or (c) can undergo a specialization step which preserves annotations. Moreover, each specialization step $w \longmapsto_s w'$ maintains Λ_{ml} convertibility on the corresponding erasures (i.e., $\lfloor w \rfloor =_{ml} \lfloor w' \rfloor$).

Lemma 2. (specialization properties)

— If $\Gamma_d \vdash_{bt} e : \mathsf{nat}[w : \mathsf{s}]$ then $w \equiv \ulcorner n \urcorner \equiv e$ for some number n.

— If $\Gamma_d \vdash_{bt} e : \tau_1 \rightarrow \tau_2[w : \varphi_1 \rightarrow \varphi_2]$ then $w \equiv \lambda x\,.\,w'$, $e \equiv \lambda x\,.\,\lfloor w' \rfloor$ and $\Gamma_d, x : \tau_1[\varphi_1] \vdash_{bt} \lfloor w' \rfloor : \tau_2[w' : \varphi_2]$ for some $w' \in \mathit{Terms}[\Lambda_{ml}^{bt}]$.

— If $\Gamma_d \vdash_{bt} e : \widetilde{\tau}[w : \widetilde{\varphi}]$ then exactly one of the following statements holds:

Interpretation rules:

$$
\begin{array}{rcl@{\qquad}rcl}
\mathsf{succ}\,\ulcorner n \urcorner & \longmapsto_i & \mathsf{unit}\,\ulcorner n+1 \urcorner & \mathsf{pred}\,\ulcorner n+1 \urcorner & \longmapsto_i & \mathsf{unit}\,\ulcorner n \urcorner \\
(\lambda x\,.\,w_0)\,@\,w_1 & \longmapsto_i & w_0[x := w_1] & \mathsf{pred}\,\ulcorner 0 \urcorner & \longmapsto_i & \mathsf{unit}\,\ulcorner 0 \urcorner \\
\mathsf{fix}\,x.\,w & \longmapsto_i & w[x := \mathsf{fix}\,x.\,w] & \mathsf{if0}\,\ulcorner 0 \urcorner\,w_2\,w_3 & \longmapsto_i & w_2 \\
\mathsf{let}\;x \Leftarrow \mathsf{unit}\,w_1\;\mathsf{in}\;w_2 & \longmapsto_i & w_2[x := w_1] & \mathsf{if0}\,\ulcorner n+1 \urcorner\,w_2\,w_3 & \longmapsto_i & w_3
\end{array}
$$

$$
\frac{w \longmapsto_i w'}{E_i[w] \longmapsto_i E_i[w']} \qquad E_i \;::=\; \mathsf{let}\;x \Leftarrow [\cdot]\;\mathsf{in}\;w_2
$$

$$
\frac{w \longmapsto_i w'}{w \longmapsto_s w'}
$$

Compilation rules:

$$
\mathsf{lift}\,\ulcorner n \urcorner \longmapsto_s \underline{\ulcorner n \urcorner} \qquad \frac{w \longmapsto_s w'}{E_s[w] \longmapsto_s E_s[w']}
$$

$$
\begin{array}{rcl}
E_s & ::= & \underline{\mathsf{pred}}\,[\cdot] \;\mid\; \underline{\mathsf{succ}}\,[\cdot] \;\mid\; \underline{\mathsf{if0}}\,[\cdot]\,w_2\,w_3 \;\mid\; \underline{\mathsf{if0}}\,r_1\,[\cdot]\,w_3 \;\mid\; \underline{\mathsf{if0}}\,r_1\,r_2\,[\cdot] \;\mid \\
 & & \underline{\lambda} x\,.\,[\cdot] \;\mid\; [\cdot]\,\underline{@}\,w_1 \;\mid\; r_0\,\underline{@}\,[\cdot] \;\mid\; \underline{\mathsf{fix}}\,x.\,[\cdot] \;\mid \\
 & & \underline{\mathsf{unit}}\,[\cdot] \;\mid\; \underline{\mathsf{let}}\;x \Leftarrow [\cdot]\;\underline{\mathsf{in}}\;w_2 \;\mid\; \underline{\mathsf{let}}\;x \Leftarrow r_1\;\underline{\mathsf{in}}\;[\cdot] \quad \textit{...where } r_i \in \textit{Residual-terms}[\Lambda^{bt}_{ml}]
\end{array}
$$

Fig. 8. Specialization rules for Λ^{bt}_{ml}

1 $w \equiv \mathsf{unit}\,w'$, $e \equiv \mathsf{unit}\,\lfloor w' \rfloor$ and $\Gamma_d \vdash_{bt} \lfloor w' \rfloor : \tau[w' : \varphi]$ for some $w' \in \textit{Terms}[\Lambda^{bt}_{ml}]$, or

2 $w \longmapsto_i w'$ and $\Gamma_d \vdash_{bt} \lfloor w' \rfloor : \widetilde{\tau}[w' : \widetilde{\varphi}]$ and $e \equiv \lfloor w \rfloor \twoheadrightarrow_{ml} \lfloor w' \rfloor$.

— If $\Gamma_d \vdash_{bt} e : \mathsf{nat}[w : \mathsf{d}]$ then exactly one of the following statements holds:

1 $w \in \textit{Residual-terms}[\Lambda^{bt}_{ml}]$, or

2 $w \longmapsto_s w'$, $\Gamma_d \vdash_{bt} \lfloor w' \rfloor : \tau[w' : \mathsf{d}]$, and $e \equiv \lfloor w' \rfloor \equiv \lfloor w \rfloor \equiv \ulcorner n \urcorner$ for some number n.

— If $\Gamma_d \vdash_{bt} e : \tau_1 \to \tau_2[w : \mathsf{d}]$ then exactly one of the following statements holds:

1 $w \in \textit{Residual-terms}[\Lambda^{bt}_{ml}]$, or

2 $w \longmapsto_s w'$ and $\Gamma_d \vdash_{bt} \lfloor w' \rfloor : \tau[w' : \mathsf{d}]$ and $e \equiv \lfloor w \rfloor \twoheadrightarrow_{ml} \lfloor w' \rfloor$.

— If $\Gamma_d \vdash_{bt} e : \widetilde{\tau}[w : \mathsf{d}]$ then exactly one of the following statements holds:

1 $w \in \textit{Residual-terms}[\Lambda^{bt}_{ml}]$, or

2 $w \longmapsto_s w'$ and $\Gamma_d \vdash_{bt} \lfloor w' \rfloor : \widetilde{\tau}[w' : \mathsf{d}]$ and $e \equiv \lfloor w \rfloor \twoheadrightarrow_{ml} \lfloor w' \rfloor$.

Proof. by induction over the height of the derivation of $\Gamma_d \vdash_{bt} e : \tau[w : \varphi]$ relying on the property that if $\Gamma, x : \tau_1[\varphi_1] \vdash_{bt} e_0 : \tau_0[w_0 : \varphi_0]$ and $\Gamma \vdash_{bt} e_1 : \tau_1[w_1 : \varphi_1]$ then $\Gamma \vdash_{bt} e_0[x := e_1] : \tau_0[w_0[x := w_1] : \varphi_0]$. □

It is easy to check that if $w \longmapsto_s w'$ and $w \longmapsto_s w''$ then $w' \equiv w''$. Furthermore, Lemma 2 shows that well-annotated terms $r \in \mathit{Residual\text{-}terms}[\Lambda^{bt}_{ml}]$ cannot undergo further specialization steps. This justifies the definition of the following (partial) function in terms of the reflexive, transitive closure of $\longmapsto_s$.

Definition 5. (specializer) For $\Gamma_d \vdash_{bt} e : \widetilde{\mathsf{nat}}[w : \mathsf{d}]$,

$$\mathit{spec}\, w = r \quad \mathit{iff} \quad w \longmapsto^*_s r$$

where $r \in \mathit{Residual\text{-}terms}[\Lambda^{bt}_{ml}]$.

3.3. *Correctness of binding-time analysis and specialization*

A binding-time analysis is correct if it always produces sound directives. Directives are unsound if they direct the specializer to attempt the reduction of a non-redex. The last component of Lemma 2 implies that any binding-time analysis satisfying our constraints is correct: when an annotated term (with only dynamic free variables) is specialized, the specializer always finds a redex to contract, or terminates because only residual components are left. In other words, the specializer never "sticks" on a non-redex.

Theorem 2. (correctness of binding-time analysis) For all $\Gamma_d \vdash_{bt} e : \widetilde{\mathsf{nat}}[w : \mathsf{d}]$, exactly one of the following statements holds:

1 $\mathit{spec}\, w = r$ where $r \in \mathit{Residual\text{-}terms}[\Lambda^{bt}_{ml}]$, or
2 $\mathit{spec}\, w\!\uparrow$ and w heads an infinite sequence of specialization steps.

Proof. follows from the definition of *spec* (Definition 5) and Lemma 2. □

This statement of binding-time correctness is analogous to statements of binding-time correctness for the λ-calculus (Jones et al. 1993; Palsberg 1993; Wand 1993).

A specializer is sound if its steps reflect a meaning-preserving transformation of the source program.

Theorem 3. (soundness of specialization) For all $\Gamma_d \vdash_{bt} e : \widetilde{\mathsf{nat}}[w : \mathsf{d}]$,

$$\mathit{spec}\, w\!\downarrow \quad \mathit{implies} \quad e \approx \lfloor \mathit{spec}\, w \rfloor$$

Proof. Given the definition of *spec* (Definition 5) and Lemma 2, one has $e =_{ml} \lfloor \mathit{spec}\, w \rfloor$ by induction over the number of specialization steps. The soundness of the Λ_{ml} calculus (Theorem 1) then gives the desired result. □

3.4. *Partial evaluation*

An offline partial evaluator for Λ_{ml} is obtained by composing binding-time analysis with specialization. When supplied with a source expression e, a binding-time assumption Γ_{bt}, and a collection of "known" data σ_s, the partial evaluator specializes the program to the known data as directed by the binding-time analysis.

Definition 6. (partial evaluator) Let *bta* be a correct binding-time analysis, $\Gamma_{bt} = \Gamma_s \cup \Gamma_d$ be a binding-time assumption for $\Gamma \vdash_{ml} e : \widetilde{\mathsf{nat}}$, and σ_s be a closed substitution compatible with Γ_s, then

$$pe\; e\; \Gamma_{bt}\; \sigma_s \;\; \stackrel{\text{def}}{=} \;\; \lfloor spec\,(bta\; \Gamma_{bt}\; e)\sigma_s \rfloor$$

The correctness theorem for the partial evaluator is analogous to Kleene's S^m_n-theorem: partial evaluation computes a program specialized to known input data (σ_s) such that running the specialized program on the remaining input data (σ_d) yields a result observationally equivalent to the result of running the source program on the complete input data.

Theorem 4. (correctness of partial evaluation) Let $\Gamma_{bt} = \Gamma_s \cup \Gamma_d$ be a binding-time assumption for $\Gamma \vdash_{ml} e : \widetilde{\mathsf{nat}}$, let σ_s be a closed substitution compatible with Γ_s, and let σ_d be a closed substitution compatible with $\lfloor \Gamma_d \rfloor$. If $(pe\; e\; \Gamma_{bt}\; \sigma_s)\!\downarrow$, then

$$e\lfloor \sigma_s \rfloor \sigma_d \;\approx\; (pe\; e\; \Gamma_{bt}\; \sigma_s)\sigma_d$$

Proof. follows from Theorem 3 and the definition of specialization (Definition 5) and partial evaluation (Definition 6). $\square$

4. Call unfolding

At the core of a partial evaluator lies call unfolding. This basic transformation enables static information to be propagated across procedure boundaries. Call unfolding is implemented using the copy rule. Thus it is only sound in a call-by-name setting such as Launchbury's (Launchbury 1991). In a call-by-value setting, call unfolding is unsound (i.e., it does not preserve observational equivalence).

For example, consider the following Λ-term (under call-by-value), occurring in a static context where e_1 is dynamic.

$$\mathsf{pred}\;((\lambda x\,.\,\ulcorner 43 \urcorner)\;@\;e_1)$$

Unfolding the inner call is unsound in general because e_1 may diverge. Yet a partial evaluator such as Lambda-Mix would unfold it (Gomard 1992; Gomard and Jones 1991).

Through a systematic insertion of let-expressions, partial evaluators such as Similix or Schism ensure sound call unfolding (Bondorf and Danvy 1991; Consel 1993). They would unfold the term above into the following term

$$\mathsf{pred}\;(\text{let}\; x = e_1 \;\text{in}\; 43)$$

and would residualize the let expression to preserve the termination properties of the source term. Now a partial evaluator only needs to move the context $\mathsf{pred}\;[\cdot]$ inside the let expression to enable a static reduction that yields the expected answer. This may require a binding-time improvement, as investigated in Section 5. The $\longmapsto_x$ steps below represent how the above expression would be treated e.g., in Similix.

$$\begin{array}{lll} & \mathsf{pred}\ (\text{let } y = e_1 \text{ in } (\lambda x\,.\,43)\ @\ y) & \\ \longmapsto_x & \mathsf{pred}\ (\text{let } y = e_1 \text{ in } 43) & \textit{...unfold call} \\ \longmapsto_x & \text{let } y = e_1 \text{ in } \mathsf{pred}\ 43 & \textit{...binding-time improvement} \\ \longmapsto_x & \text{let } y = e_1 \text{ in } 42 & \textit{...static computation} \end{array}$$

In Λ_{ml}, evaluation steps only occur in let contexts and thus their result is named. Encodings of Λ into Λ_{ml} (such as those in Appendix A.2) insert let constructs at all evaluation contexts (rather than in an ad hoc manner as above). Because of the more natural placement of let expressions, the binding-time improvement step in the evaluation above is avoided — specialization steps are simply reductions in the Λ_{ml} calculus.

For example, the call-by-value encoding of Appendix A.2 yields the following term (after performing some initial *let.β* reductions).

$$\begin{array}{l} e \equiv \mathsf{let}\ y_1 \Leftarrow \mathcal{E}_v \langle\!|e_1|\!\rangle \\ \quad\ \ \mathsf{in\ let}\ y_2 \Leftarrow (\lambda x\,.\,\mathsf{unit}\ \ulcorner 43 \urcorner)\ @\ y_1 \\ \quad\quad \mathsf{in\ pred}\ y_2 \end{array}$$

Assuming that e occurs in a static context and that $\mathcal{E}_v \langle\!|e_1|\!\rangle$ is dynamic, binding-time analysis would associate e with the following annotated term (where w_1 is the annotation of $\mathcal{E}_v \langle\!|e_1|\!\rangle$).

$$\begin{array}{l} w \equiv \underline{\mathsf{let}}\ y_1 \Leftarrow w_1 \\ \quad\ \ \underline{\mathsf{in}}\ \mathsf{let}\ y_2 \Leftarrow (\lambda x\,.\,\mathsf{unit}\ \ulcorner 43 \urcorner)\ @\ y_1 \\ \quad\quad \mathsf{in\ pred}\ y_2 \end{array}$$

Specialization reduces the inner let expression and yield the expected answer.

$$\begin{array}{ll} & w \\ \longmapsto_s & \underline{\mathsf{let}}\ y_1 \Leftarrow \mathcal{E}_v \langle\!|e|\!\rangle\ \underline{\mathsf{in}}\ \mathsf{let}\ y_2 \Leftarrow \mathsf{unit}\ \ulcorner 43 \urcorner\ \mathsf{in\ pred}\ y_2 \\ \longmapsto_s & \underline{\mathsf{let}}\ y_1 \Leftarrow \mathcal{E}_v \langle\!|e|\!\rangle\ \underline{\mathsf{in}}\ \mathsf{pred}\ 43 \\ \longmapsto_s & \mathsf{let}\ y_1 \Leftarrow \mathcal{E}_v \langle\!|e|\!\rangle\ \mathsf{in\ unit}\ 42 \end{array}$$

Finally, another advantage of phrasing partial evaluation in terms of Λ_{ml} is that concepts of proper let-insertion and binding-time improvements (as discussed in the following section) can be presented independently of evaluation order. For example, our formalization of these concepts still remains valid if one considers e.g., a mixed evaluation strategy (Danvy and Hatcliff 1993).

5. Control-based binding-time improvements

The partial evaluator defined in Section 3 is sensitive to the *structure* of source programs. Consider the two following Λ_{ml}-convertible expressions.

$$\begin{array}{lll} e_1 & \equiv & \mathsf{let}\ v_1 \Leftarrow (\mathsf{let}\ v_2 \Leftarrow x_2\ \mathsf{in\ succ}\ x_1)\ \mathsf{in}\ e \qquad \textit{...where } v_2 \notin FV(e) \\ e_2 & \equiv & \mathsf{let}\ v_2 \Leftarrow x_2\ \mathsf{in\ let}\ v_1 \Leftarrow \mathsf{succ}\ x_1\ \mathsf{in}\ e \end{array}$$

Assume $\Gamma, x_1 : \mathsf{nat}, x_2 : \widetilde{\mathsf{nat}} \vdash_{ml} e_1, e_2 : \widetilde{\tau}$, and let $\Gamma_{bt} = \Gamma'_{bt}, x_1 : \mathsf{nat}[\mathsf{s}], x_2 : \widetilde{\mathsf{nat}}[\mathsf{d}]$ be binding-time assumptions for e_1 and e_2 (i.e., x_1 is a static variable, x_2 is a dynamic variable).

Given the binding-time rules of Figures 6 and 7, e_1 must map to the following expression w_1 for some w (i.e., $\Gamma \vdash_{bt} e_1 : \widetilde{\tau}[w_1 : \mathsf{d}]$ is derivable).

$$w_1 \;\equiv\; \underline{\mathsf{let}}\, v_1 \Leftarrow (\underline{\mathsf{let}}\, v_2 \Leftarrow x_2\, \underline{\mathsf{in}}\, \underline{\mathsf{succ}}\, \mathsf{lift}\, x_1)\, \underline{\mathsf{in}}\, w$$

On the other hand, e_2 may map to the following expression w_2 for some w' (i.e., $\Gamma \vdash_{bt} e_2 : \widetilde{\tau}[w_2 : \mathsf{d}]$ is derivable).

$$w_2 \;\equiv\; \underline{\mathsf{let}}\, v_2 \Leftarrow x_2\, \underline{\mathsf{in}}\, \mathsf{let}\, v_1 \Leftarrow \mathsf{succ}\, x_1\, \mathsf{in}\, w'$$

In w_1, the let associated with v_1 is forced to be residual. In w_2, it can be eliminable — which may enable further static computations, particularly within a loop.

This example captures in a nutshell the need for *binding-time improvements*: meaning-preserving transformations over source programs that enable more source expressions to be classified as static (and thus to be computed away statically, which yields more efficient specialized programs). In this example, e_1 can be "binding-time improved" by transforming it to e_2 using the *let.assoc* reduction of Λ_{ml} calculus (see Figure 3).

5.1. *Flattening before binding-time analysis*

The example above illustrates that the *let.assoc* reduction gives a useful binding-time improvement. This (along with the fact that *let.assoc* is confluent and strongly normalizing (Hatcliff 1994)) suggests a general strategy for improving binding times: before analyzing the binding times of a program, map this program to its *let.assoc* normal form.

The following example, however, illustrates that this strategy does not discover all binding-time improvements associated with the *let.assoc* reduction. *During specialization*, unfolding a call may expose flattening opportunities that are not apparent in the source program.

For example, assume that the following variant of e_1 above is in *let.assoc* normal form.

$$e_3 \;\equiv\; \mathsf{let}\, v_1 \Leftarrow ((\lambda z\, .\, \mathsf{let}\, v_2 \Leftarrow x_2\, \mathsf{in}\, \mathsf{succ}\, x_1)\, @\, \ulcorner 0 \urcorner)\, \mathsf{in}\, e \qquad \ldots \textit{where } v_2 \notin FV(e)$$

Using the same binding-time assumptions Γ_{bt} as above, e_3 maps to w_3 below for some w'' (i.e., the let associated with v_2 is forced to be residual).

$$w_3 \;\equiv\; \underline{\mathsf{let}}\, v_1 \Leftarrow ((\lambda z\, .\, \underline{\mathsf{let}}\, v_2 \Leftarrow x_2\, \underline{\mathsf{in}}\, \underline{\mathsf{succ}}\, \mathsf{lift}\, x_1)\, @\, \ulcorner 0 \urcorner)\, \underline{\mathsf{in}}\, w''$$

Specialization (after supply static data $\ulcorner 42 \urcorner$ for x_1 includes the following step.

$$\begin{array}{ll} & \underline{\mathsf{let}}\, v_1 \Leftarrow ((\lambda z\, .\, \underline{\mathsf{let}}\, v_2 \Leftarrow x_2\, \underline{\mathsf{in}}\, \underline{\mathsf{succ}}\, \mathsf{lift}\, \ulcorner 42 \urcorner)\, @\, \ulcorner 0 \urcorner)\, \underline{\mathsf{in}}\, w'' \\ \longmapsto_s & \underline{\mathsf{let}}\, v_1 \Leftarrow (\underline{\mathsf{let}}\, v_2 \Leftarrow x_2\, \underline{\mathsf{in}}\, \underline{\mathsf{succ}}\, \mathsf{lift}\, \ulcorner 42 \urcorner)\, \underline{\mathsf{in}}\, w'' \end{array}$$

Note that this last term has the same problematic form as w_1 above.

5.2. *Flattening during specialization*

The solution is to incorporate the flattening rule *let.assoc* of Figure 3 as a specialization step. One might be tempted to simply redefine specialization so that the specialization of w_3 above proceeds as follows (i.e., a flattening step is taken whenever possible).

$$\begin{array}{ll} & \underline{\mathsf{let}}\, v_1 \Leftarrow ((\lambda z\, .\, \underline{\mathsf{let}}\, v_2 \Leftarrow x_2\, \underline{\mathsf{in}}\, \underline{\mathsf{succ}}\, \mathsf{lift}\, \ulcorner 42 \urcorner) \,@\, \ulcorner 0 \urcorner)\, \underline{\mathsf{in}}\, w'' \\ \longmapsto_s & \underline{\mathsf{let}}\, v_1 \Leftarrow (\underline{\mathsf{let}}\, v_2 \Leftarrow x_2\, \underline{\mathsf{in}}\, \underline{\mathsf{succ}}\, \mathsf{lift}\, \ulcorner 42 \urcorner)\, \underline{\mathsf{in}}\, w'' \\ \longmapsto_f & \underline{\mathsf{let}}\, v_2 \Leftarrow x_2\, \underline{\mathsf{in}}\, \underline{\mathsf{let}}\, v_1 \Leftarrow \underline{\mathsf{succ}}\, \mathsf{lift}\, \ulcorner 42 \urcorner\, \underline{\mathsf{in}}\, w'' \end{array}$$

At this point, however, we are no better off than we were before since the specialization directives indicate that the let associated with v_1 must be residualized.

Since it is the task of binding-time analysis to direct the specializer, the binding-time analysis also must reflect the possible use of the *let.assoc* rule. In the $(*)$ rule of Figure 7, the body of the $\underline{\mathsf{let}}$ is forced to have a specialization type (i.e., binding-time) of d since it cannot be consumed during specialization. However, after adding the flatten rule as a specialization step, a static expression in the body of a $\underline{\mathsf{let}}$ can be consumed. Allowing the body of the $\underline{\mathsf{let}}$ to have a static binding-time leads to the following annotation of e_3 above.

$$w_4 \quad \equiv \quad \mathsf{let}\, v_1 \Leftarrow ((\lambda z\, .\, \underline{\mathsf{let}}\, v_2 \Leftarrow x_2\, \underline{\mathsf{in}}\, \mathsf{succ}\, x_1) \,@\, \ulcorner 0 \urcorner)\, \mathsf{in}\, w'''$$

A specializer which incorporates the flattening rule now gives the desired behaviour.

$$\begin{array}{ll} & \mathsf{let}\, v_1 \Leftarrow ((\lambda z\, .\, \underline{\mathsf{let}}\, v_2 \Leftarrow x_2\, \underline{\mathsf{in}}\, \mathsf{succ}\, x_1) \,@\, \ulcorner 0 \urcorner)\, \mathsf{in}\, w''' \\ \longmapsto_s & \mathsf{let}\, v_1 \Leftarrow (\underline{\mathsf{let}}\, v_2 \Leftarrow x_2\, \underline{\mathsf{in}}\, \mathsf{succ}\, \ulcorner 42 \urcorner)\, \mathsf{in}\, w''' \\ \longmapsto_f & \underline{\mathsf{let}}\, v_2 \Leftarrow x_2\, \underline{\mathsf{in}}\, \mathsf{let}\, v_1 \Leftarrow \mathsf{succ}\, \ulcorner 42 \urcorner\, \mathsf{in}\, w''' \\ \longmapsto_s & \underline{\mathsf{let}}\, v_2 \Leftarrow x_2\, \underline{\mathsf{in}}\, \mathsf{let}\, v_1 \Leftarrow \mathsf{unit}\, \ulcorner 43 \urcorner\, \mathsf{in}\, w''' \\ \longmapsto_s & \underline{\mathsf{let}}\, v_2 \Leftarrow x_2\, \underline{\mathsf{in}}\, w'''[v_1 := \ulcorner 43 \urcorner] \end{array}$$

We formalize the modified binding-time analysis and specializer in the following sections.

5.2.1. *Binding-time analysis* Replacing the $(*)$ rule of Figure 7 with the $(**)$ rule in Figure 9 gives improved constraints $\Gamma \vdash^{+}_{bt} e : \tau[w : \varphi]$. The following property verifies that the improved constraints are at least as "good" as the previous ones. The example above shows that in many cases the improved constraints are better, i.e., they give more opportunities for static computation.

Property 1. $\Gamma \vdash_{bt} e : \tau[w : \varphi]$ implies $\Gamma \vdash^{+}_{bt} e : \tau[w : \varphi]$

Proof. by induction over the derivation of $\Gamma \vdash_{bt} e : \tau[w : \varphi]$. □

Binding-time analysis is defined as before (Definition 4). bta^{+} denotes the analysis functions based on the improved constraints.

5.2.2. *Specialization* Figure 10 gives specialization steps which incorporate the flattening rules. The first flattening rule gives the previously discussed binding-time improvements. The second rule is added for stylistic reasons. It simplifies the presentation so that flattening does not occur in an interpreter context E_i but only in specialization contexts E_s. Essentially, the specializer will attempt to apply flattening rules first. Otherwise, specialization proceeds as before. Note that in applying the *let.assoc* reduction, a renaming may be necessary to avoid variable capture (in Figure 3, the condition is that $x_1 \notin FV(e_3)$). We assume that necessary renamings take place when using the flattening rules of Figure 10 and we omit the corresponding formalization.

$$\frac{\Gamma \vdash^{+}_{bt} e_1 : \widetilde{\tau_1}[w_1 : \mathsf{d}] \qquad \Gamma, x : \tau_1[\mathsf{d}] \vdash^{+}_{bt} e_2 : \widetilde{\tau_2}[w_2 : \varphi]}{\Gamma \vdash^{+}_{bt} \mathsf{let}\ x \Leftarrow e_1\ \mathsf{in}\ e_2 : \widetilde{\tau_2}[\underline{\mathsf{let}}\ x \Leftarrow w_1\ \underline{\mathsf{in}}\ w_2 : \varphi]}(**)$$

Fig. 9. Modified binding-time constraints for Λ^{bt}_{ml}

Flattening rules:

$$\mathsf{let}\ x_2 \Leftarrow (\underline{\mathsf{let}}\ x_1 \Leftarrow w_1\ \underline{\mathsf{in}}\ w_2)\ \mathsf{in}\ w_3 \ \longmapsto_f\ \underline{\mathsf{let}}\ x_1 \Leftarrow w_1\ \underline{\mathsf{in}}\ \mathsf{let}\ x_2 \Leftarrow w_2\ \mathsf{in}\ w_3$$

$$\mathsf{let}\ x_2 \Leftarrow (\mathsf{let}\ x_1 \Leftarrow w_1\ \mathsf{in}\ w_2)\ \mathsf{in}\ w_3 \ \longmapsto_f\ \mathsf{let}\ x_1 \Leftarrow w_1\ \mathsf{in}\ \mathsf{let}\ x_2 \Leftarrow w_2\ \mathsf{in}\ w_3$$

$$\frac{w \ \longmapsto_f\ w'}{w \ \longmapsto_{s+}\ w'}$$

Interpretation rules:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\mathsf{succ}\ulcorner n \urcorner & \longmapsto_i & \mathsf{unit}\ulcorner n+1 \urcorner & \mathsf{pred}\ulcorner n+1 \urcorner & \longmapsto_i & \mathsf{unit}\ulcorner n \urcorner \\
(\lambda x\,.\,w_0)\,@\,w_1 & \longmapsto_i & w_0[x := w_1] & \mathsf{pred}\ulcorner 0 \urcorner & \longmapsto_i & \mathsf{unit}\ulcorner 0 \urcorner \\
\mathsf{fix}\,x.\,w & \longmapsto_i & w[x := \mathsf{fix}\,x.\,w] & \mathsf{if0}\ulcorner 0 \urcorner\, w_2\, w_3 & \longmapsto_i & w_2 \\
\mathsf{let}\ x \Leftarrow \mathsf{unit}\ w_1\ \mathsf{in}\ w_2 & \longmapsto_i & w_2[x := w_1] & \mathsf{if0}\ulcorner n+1 \urcorner\, w_2\, w_3 & \longmapsto_i & w_3
\end{array}$$

$$\frac{w \ \longmapsto_i\ w'}{E_i[w] \ \longmapsto_i\ E_i[w']} \qquad\qquad E_i \quad ::= \quad \mathsf{let}\ x \Leftarrow [\cdot]\ \mathsf{in}\ w_2$$

$$\frac{w \ \longmapsto_i\ w'}{w \ \longmapsto_{s+}\ w'} \qquad \textit{where } w \not\longmapsto_f\ w''$$

Compilation rules:

$$\mathsf{lift}\ulcorner n \urcorner \ \longmapsto_{s+}\ \underline{\ulcorner n \urcorner} \qquad\qquad \frac{w \ \longmapsto_{s+}\ w'}{E_s[w] \ \longmapsto_{s+}\ E_s[w']}$$

$$\begin{array}{rcl}
E_s & ::= & \underline{\mathsf{pred}}\,[\cdot] \mid \underline{\mathsf{succ}}\,[\cdot] \mid \underline{\mathsf{if0}}\,[\cdot]\,w_2\,w_3 \mid \underline{\mathsf{if0}}\,r_1\,[\cdot]\,w_3 \mid \underline{\mathsf{if0}}\,r_1\,r_2\,[\cdot] \mid \\
 & & \underline{\lambda} x\,.\,[\cdot] \mid [\cdot]\,\underline{@}\,w_1 \mid r_0\,\underline{@}\,[\cdot] \mid \underline{\mathsf{fix}}\,x.\,[\cdot] \mid \\
 & & \underline{\mathsf{unit}}\,[\cdot] \mid \underline{\mathsf{let}}\ x \Leftarrow [\cdot]\ \underline{\mathsf{in}}\ w_2 \mid \underline{\mathsf{let}}\ x \Leftarrow r_1\ \underline{\mathsf{in}}\ [\cdot] \qquad \textit{...where } r_i \in \textit{Residual-terms}[\Lambda^{bt}_{ml}]
\end{array}$$

Fig. 10. Modified specialization rules for Λ^{bt}_{ml}

The following lemma gives specialization properties for the Λ_{ml}^{bt} terms satisfying judgements of the form $\Gamma_d \vdash_{bt}^{+} e : \tau[w : \varphi_\tau]$. It only differs from the previous specialization properties (Lemma 2) in that the third component (dealing with judgements of the form $\Gamma_d \vdash_{bt} e : \widetilde{\tau}[w : \widetilde{\varphi}]$) now reflects the improved treatment of let constructs.

Lemma 3. (specialization properties)

— If $\Gamma_d \vdash_{bt}^{+} e : \mathsf{nat}[w : \mathsf{s}]$ then $w \equiv \ulcorner n \urcorner \equiv e$ for some number n.

— If $\Gamma_d \vdash_{bt}^{+} e : \tau_1 \rightarrow \tau_2[w : \varphi_1 \rightarrow \varphi_2]$ then $w \equiv \lambda x \,.\, w'$, $e \equiv \lambda x \,.\, \lfloor w' \rfloor$ and $\Gamma_d, x : \tau_1[\varphi_1] \vdash_{bt}^{+} \lfloor w' \rfloor : \tau_2[w' : \varphi_2]$ for some $w' \in \mathit{Terms}[\Lambda_{ml}^{bt}]$.

— If $\Gamma_d \vdash_{bt}^{+} e : \widetilde{\tau}[w : \widetilde{\varphi}]$ then exactly one of the following statements holds:

1 $w \equiv \mathsf{unit}\, w'$, $e \equiv \mathsf{unit}\, \lfloor w' \rfloor$ and $\Gamma \vdash_{bt}^{+} \lfloor w' \rfloor : \tau[w' : \varphi]$ for some $w' \in \mathit{Terms}[\Lambda_{ml}^{bt}]$, or

2 $w \equiv \underline{\mathsf{let}}\, x \Leftarrow w_1 \,\underline{\mathsf{in}}\, w_2$, $e \equiv \mathsf{let}\, x \Leftarrow \lfloor w_1 \rfloor \,\mathsf{in}\, \lfloor w_2 \rfloor$, and $\Gamma_d \vdash_{bt}^{+} \lfloor w_1 \rfloor : \widetilde{\tau}_1[w_1 : \mathsf{d}]$, $\Gamma_d, x : \tau_1[\mathsf{d}] \vdash_{bt}^{+} \lfloor w_2 \rfloor : \widetilde{\tau}_2[w_2 : \widetilde{\varphi}]$, for some $w_1, w_2 \in \mathit{Terms}[\Lambda_{ml}^{bt}]$, or

3 $w \longmapsto_i w'$ and $\Gamma_d \vdash_{bt}^{+} \lfloor w' \rfloor : \widetilde{\tau}[w' : \widetilde{\varphi}]$ and $e \equiv \lfloor w \rfloor \twoheadrightarrow_{ml} \lfloor w' \rfloor$, or

4 $w \longmapsto_f w'$ and $\Gamma_d \vdash_{bt}^{+} \lfloor w' \rfloor : \widetilde{\tau}[w' : \widetilde{\varphi}]$ and $e \equiv \lfloor w \rfloor \twoheadrightarrow_{ml} \lfloor w' \rfloor$.

— If $\Gamma_d \vdash_{bt}^{+} e : \mathsf{nat}[w : \mathsf{d}]$ then exactly one of the following statements holds:

1 $w \in \mathit{Residual\text{-}terms}[\Lambda_{ml}^{bt}]$, or

2 $w \longmapsto_{s^+} w'$, $\Gamma_d \vdash_{bt}^{+} \lfloor w' \rfloor : \tau[w' : \mathsf{d}]$, and $e \equiv \lfloor w' \rfloor \equiv \lfloor w \rfloor \equiv \ulcorner n \urcorner$ for some number n.

— If $\Gamma_d \vdash_{bt}^{+} e : \tau_1 \rightarrow \tau_2[w : \mathsf{d}]$ then exactly one of the following statements holds:

1 $w \in \mathit{Residual\text{-}terms}[\Lambda_{ml}^{bt}]$, or

2 $w \longmapsto_{s^+} w'$ and $\Gamma_d \vdash_{bt}^{+} \lfloor w' \rfloor : \tau[w' : \mathsf{d}]$ and $e \equiv \lfloor w \rfloor \twoheadrightarrow_{ml} \lfloor w' \rfloor$.

— If $\Gamma_d \vdash_{bt}^{+} e : \widetilde{\tau}[w : \mathsf{d}]$ then exactly one of the following statements holds:

1 $w \in \mathit{Residual\text{-}terms}[\Lambda_{ml}^{bt}]$, or

2 $w \longmapsto_{s^+} w'$ and $\Gamma_d \vdash_{bt}^{+} \lfloor w' \rfloor : \widetilde{\tau}[w' : \mathsf{d}]$ and $e \equiv \lfloor w \rfloor \twoheadrightarrow_{ml} \lfloor w' \rfloor$.

Proof. by induction over the height of the derivation of $\Gamma_d \vdash_{bt}^{+} e : \tau[w : \varphi]$ relying on the property that if $\Gamma, x : \tau_1[\varphi_1] \vdash_{bt}^{+} e_0 : \tau_0[w_0 : \varphi_0]$ and $\Gamma \vdash_{bt}^{+} e_1 : \tau_1[w_1 : \varphi_1]$ then $\Gamma \vdash_{bt}^{+} e_0[x := e_1] : \tau_0[w_0[x := w_1] : \varphi_0]$. □

As before, if $w \longmapsto_{s^+} w'$ and $w \longmapsto_{s^+} w''$ then $w' \equiv w''$. The definitions of specialization and partial evaluation as well as the proofs of correctness of binding-time analysis, specialization, and partial evaluation proceed exactly as before.

6. Equivalence with continuation-based specialization

Not only does the incorporation of the *let.assoc* lead to improved binding times, but the resulting improved specializer captures the essence of control-based binding-time improvements as found in, e.g., the partial evaluator Similix (Bondorf and Danvy 1991). To substantiate this claim, we recast Bondorf's continuation-based specializer (Bondorf 1992) in terms of Λ_{ml}, and prove that this specializer is equivalent to the spec^{+} of Section 5.2.2.

Interpretation rules:

$$\langle\, \mathsf{unit}\, w\, ,\; nil \rangle \Downarrow \mathsf{unit}\, w$$

$$\frac{\langle\, \mathsf{unit}\, \ulcorner n+1 \urcorner\, ,\; \vec{E_i}\,\rangle \Downarrow a}{\langle\, \mathsf{succ}\, \ulcorner n \urcorner\, ,\; \vec{E_i}\,\rangle \Downarrow a} \qquad \frac{\langle\, \mathsf{unit}\, \ulcorner n \urcorner\, ,\; \vec{E_i}\,\rangle \Downarrow a}{\langle\, \mathsf{pred}\, \ulcorner n+1 \urcorner\, ,\; \vec{E_i}\,\rangle \Downarrow a} \qquad \frac{\langle\, \mathsf{unit}\, \ulcorner 0 \urcorner\, ,\; \vec{E_i}\,\rangle \Downarrow a}{\langle\, \mathsf{pred}\, \ulcorner 0 \urcorner\, ,\; \vec{E_i}\,\rangle \Downarrow a}$$

$$\frac{\langle\, w_2\, ,\; \vec{E_i}\,\rangle \Downarrow a}{\langle\, \mathsf{if0}\, \ulcorner 0 \urcorner\, w_2\, w_3\, ,\; \vec{E_i}\,\rangle \Downarrow a} \qquad \frac{\langle\, w_3\, ,\; \vec{E_i}\,\rangle \Downarrow a}{\langle\, \mathsf{if0}\, \ulcorner n+1 \urcorner\, w_2\, w_3\, ,\; \vec{E_i}\,\rangle \Downarrow a}$$

$$\frac{\langle\, w_0[x := w_1]\, ,\; \vec{E_i}\,\rangle \Downarrow a}{\langle\, (\lambda x\, .\, w_0)\, @\, w_1\, ,\; \vec{E_i}\,\rangle \Downarrow a} \qquad \frac{\langle\, w[x := \mathsf{fix}\, x.\, w]\, ,\; \vec{E_i}\,\rangle \Downarrow a}{\langle\, \mathsf{fix}\, x.\, w\, ,\; \vec{E_i}\,\rangle \Downarrow a}$$

$$\frac{\langle\, w_1\, ,\; \mathsf{let}\; x \Leftarrow \cdot\; \mathsf{in}\; w_2\; ::\; \vec{E_i}\,\rangle \Downarrow a}{\langle\, \mathsf{let}\; x \Leftarrow w_1\; \mathsf{in}\; w_2\, ,\; \vec{E_i}\,\rangle \Downarrow a}\; \textit{(Push)}$$

$$\frac{\langle\, w_2[x := w_1]\, ,\; \vec{E_i}\,\rangle \Downarrow a}{\langle\, \mathsf{unit}\, w_1\, ,\; \mathsf{let}\; x \Leftarrow \cdot\; \mathsf{in}\; w_2\; ::\; \vec{E_i}\,\rangle \Downarrow a}\; \textit{(Pop)}$$

Fig. 11. Continuation-based specializer for Λ^{bt}_{ml} (part 1)

Figures 11 and 12 present the continuation-based specializer as a big-step operational semantics. Bondorf's specializer is expressed denotationally, but there are standard techniques for going from a denotational specification to an operational one (Nielson and Nielson 1992b).[†] The specializer manipulates configurations of the form $\langle\, w\, ,\; \vec{E_i}\,\rangle$ where $w \in \mathit{Terms}[\Lambda^{bt}_{ml}]$ and $\vec{E_i}$ is a stack of interpretation contexts (i.e., $\vec{E_i}$ is a continuation). The intuition is that $\vec{E_i}$ accumulates static contexts (via the *(Push)* rule) and if possible moves them across dynamic let expressions to consume static values in the let body (via the *(Assoc)* rule). The *(Pop)* rule consumes a static value by reducing the top-most let in $\vec{E_i}$. In dynamic contexts (e.g., in the compilation rules), there is no need to accumulate contexts (i.e., the stack is delimited to be *nil* — which corresponds to the identity continuation).

[†] For stylistic reasons we have deviated slightly from Bondorf's presentation, where all the rules for dynamic constructs are expressed in continuation-passing style, and dynamic contexts are thus accumulated as well. This accumulation, however, is not necessary since continuations are only used to move static contexts across dynamic **let** expressions. Based on this observation (made during our initial work, in the Fall of 1994), we have written our rules so that components of dynamic constructs (e.g., $\underline{\mathsf{if0}}$) are specialized with respect to the identity continuation (i.e., *nil*).

Compilation rules:

$$\langle \underline{\ulcorner n \urcorner}\,,\ nil \rangle \Downarrow \underline{\ulcorner n \urcorner} \qquad \langle x\,,\ nil \rangle \Downarrow x \qquad \langle \mathsf{lift}\, \ulcorner n \urcorner\,,\ nil \rangle \Downarrow \underline{\ulcorner n \urcorner} \qquad \frac{\langle w\,,\ nil \rangle \Downarrow r}{\langle \underline{\mathsf{unit}}\, w\,,\ nil \rangle \Downarrow \underline{\mathsf{unit}}\, r}$$

$$\frac{\langle w\,,\ nil \rangle \Downarrow r}{\langle \underline{\mathsf{succ}}\, w\,,\ nil \rangle \Downarrow \underline{\mathsf{succ}}\, r} \qquad \frac{\langle w\,,\ nil \rangle \Downarrow r}{\langle \underline{\mathsf{pred}}\, w\,,\ nil \rangle \Downarrow \underline{\mathsf{pred}}\, r}$$

$$\frac{\langle w\,,\ nil \rangle \Downarrow r}{\langle \underline{\lambda} x\,.\, w\,,\ nil \rangle \Downarrow \underline{\lambda} x\,.\, r} \qquad \frac{\langle w\,,\ nil \rangle \Downarrow r}{\langle \underline{\mathsf{fix}}\, x\,.\, w\,,\ nil \rangle \Downarrow \underline{\mathsf{fix}}\, x\,.\, r}$$

$$\frac{\langle w_1\,,\ nil \rangle \Downarrow r_1 \qquad \langle w_2\,,\ nil \rangle \Downarrow r_2}{\langle w_1 \,\underline{@}\, w_2\,,\ nil \rangle \Downarrow r_1 \,\underline{@}\, r_2}$$

$$\frac{\langle w_1\,,\ nil \rangle \Downarrow r_1 \qquad \langle w_2\,,\ nil \rangle \Downarrow r_2 \qquad \langle w_3\,,\ nil \rangle \Downarrow r_3}{\langle \underline{\mathsf{if0}}\, w_1\ w_2\ w_3\,,\ nil \rangle \Downarrow \underline{\mathsf{if0}}\, r_1\ r_2\ r_3}$$

$$\frac{\langle w_1\,,\ nil \rangle \Downarrow r_1 \qquad \langle w_2\,,\ \vec{E_i} \rangle \Downarrow r_2}{\langle \underline{\mathsf{let}}\, x \Leftarrow w_1\ \underline{\mathsf{in}}\ w_2\,,\ \vec{E_i} \rangle \Downarrow \underline{\mathsf{let}}\, x \Leftarrow r_1\ \underline{\mathsf{in}}\ r_2}\ \mathit{(Assoc)}$$

Fig. 12. Continuation-based specializer for Λ^{bt}_{ml} (part 2)

The following theorem establishes the correspondence between the continuation-based specializer and $spec^{+}$ of Section 5.2.2.

Theorem 5. For $\Gamma_d \vdash^{+}_{bt} e : \widetilde{\mathsf{nat}}[w : \mathsf{d}]$,

$$\langle w\,,\ nil \rangle \Downarrow r \quad \text{iff} \quad w \longmapsto^{*}_{s^{+}} r$$

Proof. The proof is straightforward and follows the standard pattern for relating a big-step and small-step operational semantics (Gunter 1992, p. 111). However, one must strengthen the statement to prove so as to handle arbitrary stacks of interpretation contexts in the inductive hypothesis. We refer the reader to the extended version of this paper for the full proof (Hatcliff and Danvy 1996). □

Bondorf's presentation (Bondorf 1992) did not include a specification of binding-time analysis. However, our connection of continuation-based specialization and $spec^{+}$ shows that the constraints given by $\Gamma \vdash^{+}_{bt} e : \tau[w : \varphi]$ judgements are the proper ones (appealing to the correctness of binding-time analysis for $spec^{+}$).

7. Related work

7.1. *Computational meta-languages*

Moggi introduced his computational metalanguage as a convenient format to specify denotational semantics modularly (Moggi 1991). This metalanguage has been used for a variety of purposes including treating computational effects in functional languages (Gordon 1993; Filinski 1994; Liang et al. 1995; Steele 1994; Wadler 1992a), staging denotational-semantics definitions (Crole and Gordon 1993) and compiler translations (Benton 1995; Hatcliff and Danvy 1994; Sabry and Wadler 1996), and explaining relationships between various constructive logics (Benton et al. 1995). In an earlier work (Hatcliff 1994; Hatcliff and Danvy 1994), we have used this metalanguage to formalize the structure of continuation-passing styles.

Inspired by the present work, Nielsen has recently developed an evaluation-order independent presentation of partial evaluation and deforestation using Moggi's metalanguage (Nielsen 1997). The distinction between values and computations given by the metalanguage type system (upon which we have relied heavily) lies at the foundation of his work as well. This continues a trend of unifying various program-specialization techniques (Sørensen et al. 1994). Clearly Λ_{ml} stands as a promising testbed for this unification.

As noted above, the metalanguage and monads are commonly used to structure functional programs with computational effects. Both Nielsen (Nielsen 1997) and Dussart and Thiemann (Dussart and Thiemann 1996) use monads to structure I/O and state in the partial evaluator itself.

Particular aspects of Moggi's metalanguage highlighted in the present work also appear in other metalanguages. For example, Talcott uses let-expressions to specify computational steps (Talcott 1992). Sabry and Felleisen demonstrate how a let-based intermediate language (called *A-normal forms*) can give benefits similar to CPS in dataflow analysis (Sabry and Felleisen 1993; Sabry and Felleisen 1994).

Moggi also gave a *computational λ-calculus* λ_c. λ_c is essentially a call-by-value λ-calculus extended with equations that capture program equivalences holding under arbitrary monadic effects. It is straightforward to adapt our evaluation-order independent presentation based on Λ_{ml} to a call-by-value presentation based on λ_c. This alternative presentation expresses control-based binding-time improvements via the monadic laws as formulated in λ_c. The reassociation of let expressions is again a prominent feature. In fact, the λ_c presentation has strong connections with the work of Flanagan et al. (Flanagan et al. 1994) on the essence of compiling with continuations.

7.2. *Partial evaluation*

7.2.1. *Styles of specification* The earliest work on partial evaluation as a two-phase process specified binding-time analysis as an abstract interpretation (Jones et al. 1989). Since the work of Jones and Gomard (Gomard 1992; Gomard and Jones 1991), binding-time analysis is more often specified using type systems (we have followed this approach in the present work). Palsberg (Palsberg 1993) and Wand (Wand 1993) further clarify the rôle of such specifications in their work on the correctness of binding-time analysis.

Specializers have mostly been specified as symbolic interpreters in functional style (Jones et al. 1993). We note, however, a recent trend (including the present work) to use operational semantics for specifying specializers (Andersen 1994; Danvy et al. 1996; Hannan and Miller 1989; Sørensen et al. 1994; Sands 1995). In fact, the first author has shown that by emphasizing the logical character of type-based and operational semantics specifications, the correctness of a partial evaluator can be mechanically verified (Hatcliff 1995).

Davies and Pfenning have developed a language for expressing staged computation based on the intuistionistic modal logic S4 (Davies and Pfenning 1996; Davies 1996). The type system of this language is strikingly similar to that of Moggi's computational metalanguage. A modality analogous to the $\tilde{\cdot}$ construction of Moggi is used to type *code* objects (in our terminology, objects whose specialization type is *dynamic*). Their language also includes a let construct which can act as an "eval function". However, even though types and terms of both languages are quite similar, the similarity is superficial in the context of our application of the metalanguage. In our setting, staging information is not represented using the modality (as with Davies and Pfenning). Instead it is represented using *specialization types* which are external to the types of the meta-language itself. In our case, the modality is used to distinguish *values* from *computations* — a distinction not captured in Davies and Pfenning's work. Benton, Bierman, and de Paiva (Benton et al. 1995) flesh out connections between Moggi's computational metalanguage and the modal logic S4 in a more general context.

7.2.2. *Call unfolding and let insertion* Most specializers ensuring sound call-unfolding under call-by-value adopt the technique of let insertion, as discussed in Section 4. One may, however, also enforce soundness by simply not unfolding calls where the argument expression is dynamic, at the price of reducing specialization. This must be expressed in the binding-time analysis by forbidding binding-times such as $\mathsf{d} \rightarrow \mathsf{s}$ (i.e., a static function mapping a dynamic argument into a static result). This restriction in fact also occurs in Nielson and Nielson's two-level λ-calculus (Nielson and Nielson 1992a): the co-domain of a static function should be at least as dynamic as the domain of this function.

In contrast, consider a partial evaluator (e.g., Similix) that (1) ensures sound call-unfolding by let insertion, and (2) performs binding-time improvements by relocating static evaluation contexts inside dynamic let expressions. This partial evaluator does not constrain the domain and the codomain of static (call-by-value) functions. For example, in the term

$$((\lambda x\,.\,2) @ d) + 1$$

where d denotes a dynamic integer, the λ-abstraction is classified as a static function mapping a dynamic integer into a static integer (i.e., its binding time is $\mathsf{d} \rightarrow \mathsf{s}$). As a corollary, the addition is classified as static. The residual program reads

$$\mathsf{let}\; x \Leftarrow d \;\mathsf{in}\; 3.$$

In both techniques above, the possible binding times (i.e., specialization types) are tied to the strategy used to enforce sound call unfolding. A pleasant feature of phrasing partial evaluation in terms of Λ_{ml} is that the characterization of sound call unfolding and possible binding times are orthogonal. The distinction between values and computations in the type system means that static functions with specialization types such as $\mathsf{d} \rightarrow \mathsf{s}$ can *always* be dealt with in a sound manner. It is the encoding into Λ_{ml} where one adopts a technique for a particular evaluation order.

7.2.3. *Binding-time improvements* Consel and Danvy observed that a source transformation into continuation-passing style prevented a class of loss of static information across procedures, and they provided a syntactic characterization of this class (Consel and Danvy 1991). Holst and Gomard observed that part of the same effect (intra-procedural and insensitive to call unfolding) could be obtained by "flattening" each source procedure (Holst and Gomard 1991). Bondorf, Danvy, and Lawall observed that a further part of the same effect (accounting for call unfolding, but not crossing specialization points) could be obtained by specific control operations in the specializer itself (Bondorf 1992; Lawall and Danvy 1994), rather than by CPS-transforming the program before partial evaluation. Our framework achieves this effect by extending the operational semantics of specialization with the *let.assoc* rule. This development matches contemporary work on determining the effect of the CPS transformation on flow analysis (Nielson 1982; Sabry and Felleisen 1994): enriching a direct-style calculus can yield analyses with an added precision that matches the extra precision obtained by the relocation of contexts performed by the CPS transformation.

Section 6 shows how the theory of monads captures the essence of control-based binding-time improvements. This has practical benefits as well — it allows one to avoid using a functional representation of continuations in the specializer (one need only use the *let.assoc* rule). The disadvantage of representing continuations as functions shows up in a self-applicable partial evaluator, as self-application generates programs with many higher-order functions. Such programs are more difficult to reason about, e.g., when searching for binding-time improvements. This difficulty motivated Lawall and Danvy to stick to direct style (Lawall and Danvy 1994), and Glück and Jørgensen to devise a *multi-level cogen* (Glück and Jørgensen 1995). Thiemann has united both lines of work (Thiemann 1996). In the present case, and since continuation-passing style can be obtained from monadic style by simply selecting (a term representation of) the continuation monad, our method provides sound guidelines for treating continuations in an offline partial evaluator.

We also expect our technique to be particularly useful in online self-applicable program specializers (e.g., supercompilers). In fact, preliminary work by the first author and Glück on online self-application uses a language called *Sgraph* (Glück and Klimov 1993) where let-bindings (in the style of Λ_{ml}) are a central feature (Hatcliff and Glück 1996).

Let us briefly attempt a taxonomy of continuation-based partial evaluators. The earliest one is reported in the literature by Bondorf (Bondorf 1992). Its goal is precisely to relocate static contexts across dynamic let expressions, as in the CPS transformation. The particular brand of continuation-passing style used for this relocation is expressible in direct style, using control operators (again as in the CPS transformation). In fact, this direct-style specializer turns out to be more efficient in practice (Lawall and Danvy 1994). A parallel development is taking place in the "cogen" approach to partial evaluation (Bondorf and Dussart 1994; Lawall and Danvy 1995; Thiemann 1996). All these continuation-based specializers are specified as symbolic interpreters in functional style. An increasing number of specializers, however, are specified operationally. We have presented here such a specializer, which is continuation-based and delimits control to static contexts. This novel feature can be observed to be rippling back into the world of functional specializers, both with explicit continuations and with implicit continuations and the associated control operators (Dussart 1997).

As for data-flow binding-time improvements, they arise from insufficient binding-time coercions (Danvy et al. 1995; Danvy et al. 1996). In particular, the binding-time improvement arising from the presence of booleans and disjoint sums is known as "The Trick" (Danvy et al. 1996; Jones et al. 1993). Because of the nature of disjoint sums, this coercion takes the form of a control-based binding-time improvement: The Trick amounts to duplicating static contexts across dynamic case expressions. Again, continuations can be used to move static contexts across dynamic conditional expressions, duplicating them in the conditional branches. This transformation, however, can also be naturally accomplished using the computational meta-language (Hatcliff and Danvy 1994). It gives rise to an analogue of Figure 9 for $\mathsf{if0}$, and to a context duplication in Figure 12. It is interesting to note (this observation is due to Malmkjær) that if we consider a let expression as a "unary" case expression (i.e., a case expression with one conditional branch), then our let-rearranging rules coincide with the case-rearranging rules (also known as *commuting conversions*) that can be found both in partial evaluation (Danvy et al. 1996), program extraction (Paulin-Mohring and Werner 1993), and natural-deduction proof theory (Girard et al. 1989; Prawitz 1965).

7.2.4. *Evaluation-order independence* We have formulated control-based binding-time improvements via monads using the computational meta-language because it allows an evaluation-order independent view of binding-time analysis and specialization. This appears particularly useful in settings where adopting mixed evaluation strategies (e.g., call-by-name and call-by-value) can be employed to enhance efficiency (Danvy and Hatcliff 1993). In addition, Nielsen and Sørensen have identified situations where transforming sections of call-by-value programs using the call-by-name CPS transformation can increase specialization (Nielsen and Sørensen 1995). The evaluation-order independent partial-evaluation strategy that we have given here seems well-suited for this endeavour since Λ_{ml} allows one to encode mixed evaluation strategies while remaining in direct style (Hatcliff 1994).

$$
\begin{array}{lcl}
e & \in & \mathit{Terms}[\Lambda] \\
e & ::= & x \mid f \mid \ulcorner n \urcorner \mid \mathsf{succ}\, e \mid \mathsf{pred}\, e \mid \mathsf{if0}\, e_1\, e_2\, e_3 \mid \lambda x\!:\!\tau\,.\, e \mid e_0 @ e_1 \mid \mathsf{fix}\, f\!:\!\tau.\, e \\
 & & \\
x & \in & \mathit{Identifiers}[\Lambda] \\
f & \in & \mathit{RecIdentifiers}[\Lambda] \\
 & & \\
\tau & \in & \mathit{Types}[\Lambda] \\
\tau & ::= & \mathsf{nat} \mid \tau_1 \rightarrow \tau_2 \\
 & & \\
\Gamma & \in & \mathit{Assumptions}[\Lambda] \\
\Gamma & ::= & \cdot \mid \Gamma, x\!:\!\tau \mid \Gamma, f\!:\!\tau
\end{array}
$$

Fig. 13. The language Λ

8. Conclusion

Partial evaluation offers a practical way of staging the execution of programs in order to adapt them to the context of their execution. We have identified Moggi's computational metalanguage as a useful intermediate language for formalizing it. As a first step, we have formalized binding-time analysis and program specialization. Then we have shown how the intermediate language captures the essence of control-based binding-time improvements. Other work gives evidence that the metalanguage can provide an avenue for (a) incorporating other computational effects into partial evaluation such as I/O and state, and (b) unifying various program-specialization techniques.

Acknowledgements

Thanks are due to Karoline Malmkjær for comments on an earlier version of this paper, and to the anonymous referees for their perceptive reviews.

Most of this work was carried out while the first author was visiting the University of Copenhagen (DIKU), and we thank the DIKU TOPPS group for their interaction, encouragement, and financial support, e.g., to attend LDPL'95 in Darmstadt, Germany.

Appendix A. The Language Λ

A.1. *Syntax*

Figure 13 presents the syntax of a PCF-like language Λ. We omit the usual typing rules as well as the call-by-name and call-by-value operational semantics for Λ.

The syntax of Λ includes two categories of identifiers: *Identifiers* are used in λ-bindings; *RecIdentifiers* are used in fix-bindings. This distinction is necessary under call-by-value evaluation of Λ, where *Identifiers* will only bind to canonical terms (i.e., values) whereas

$$
\begin{array}{rcl}
\mathcal{E}_n\langle\!\!\langle \cdot \rangle\!\!\rangle & : & \Lambda \rightarrow \Lambda_{ml} \\
\mathcal{E}_n\langle\!\!\langle v \rangle\!\!\rangle & = & \mathsf{unit}\ \mathcal{E}_n\langle v \rangle \\
\mathcal{E}_n\langle\!\!\langle x \rangle\!\!\rangle & = & x \\
\mathcal{E}_n\langle\!\!\langle f \rangle\!\!\rangle & = & f \\
\mathcal{E}_n\langle\!\!\langle \mathsf{succ}\ e \rangle\!\!\rangle & = & \mathsf{let}\ y \Leftarrow \mathcal{E}_n\langle\!\!\langle e \rangle\!\!\rangle\ \mathsf{in}\ \mathsf{succ}\ y \\
\mathcal{E}_n\langle\!\!\langle \mathsf{pred}\ e \rangle\!\!\rangle & = & \mathsf{let}\ y \Leftarrow \mathcal{E}_n\langle\!\!\langle e \rangle\!\!\rangle\ \mathsf{in}\ \mathsf{pred}\ y \\
\mathcal{E}_n\langle\!\!\langle \mathsf{if0}\ e_0\ e_1\ e_2 \rangle\!\!\rangle & = & \mathsf{let}\ y_0 \Leftarrow \mathcal{E}_n\langle\!\!\langle e_0 \rangle\!\!\rangle\ \mathsf{in}\ \mathsf{if0}\ y_0\ \mathcal{E}_n\langle\!\!\langle e_1 \rangle\!\!\rangle\ \mathcal{E}_n\langle\!\!\langle e_2 \rangle\!\!\rangle \\
\mathcal{E}_n\langle\!\!\langle e_0\ @\ e_1 \rangle\!\!\rangle & = & \mathsf{let}\ y_0 \Leftarrow \mathcal{E}_n\langle\!\!\langle e_0 \rangle\!\!\rangle\ \mathsf{in}\ y_0\ @\ \mathcal{E}_n\langle\!\!\langle e_1 \rangle\!\!\rangle \\
\mathcal{E}_n\langle\!\!\langle \mathsf{fix}\, f.\, e \rangle\!\!\rangle & = & \mathsf{fix}\, f.\, \mathcal{E}_n\langle\!\!\langle e \rangle\!\!\rangle \\
\\
\mathcal{E}_n\langle \cdot \rangle & : & \mathit{Values}_n[\Lambda] \rightarrow \Lambda_{ml} \\
\mathcal{E}_n\langle \ulcorner n \urcorner \rangle & = & \ulcorner n \urcorner \\
\mathcal{E}_n\langle \lambda x\,.\,e \rangle & = & \lambda x\,.\,\mathcal{E}_n\langle\!\!\langle e \rangle\!\!\rangle \\
\\
\mathcal{E}_n\langle \mathsf{nat} \rangle & = & \mathsf{nat} \\
\mathcal{E}_n\langle \tau_1 \rightarrow \tau_2 \rangle & = & \mathcal{E}_n\langle\!\!\langle \tau_1 \rangle\!\!\rangle \rightarrow \mathcal{E}_n\langle\!\!\langle \tau_2 \rangle\!\!\rangle \\
\mathcal{E}_n\langle\!\!\langle \tau \rangle\!\!\rangle & = & \widetilde{\mathcal{E}_n\langle \tau \rangle} \\
\\
\mathcal{E}_n\langle\!\!\langle \Gamma, x : \tau \rangle\!\!\rangle & = & \mathcal{E}_n\langle\!\!\langle \Gamma \rangle\!\!\rangle, x : \mathcal{E}_n\langle\!\!\langle \tau \rangle\!\!\rangle \\
\mathcal{E}_n\langle\!\!\langle \Gamma, f : \tau \rangle\!\!\rangle & = & \mathcal{E}_n\langle\!\!\langle \Gamma \rangle\!\!\rangle, f : \mathcal{E}_n\langle\!\!\langle \tau \rangle\!\!\rangle
\end{array}
$$

Fig. 14. Call-by-name encoding $\mathcal{E}_n$ into Λ_{ml}

RecIdentifiers may bind to non-canonical terms (i.e., computations). This is reflected in the formal definition of values below.

Definition 7. (Values)

$$
\begin{array}{rclcrcl}
v & \in & \mathit{Values}_n[\Lambda] & \qquad & v & \in & \mathit{Values}_v[\Lambda] \\
v & ::= & \ulcorner n \urcorner \mid \lambda x\,.\,e & & v & ::= & x \mid \ulcorner n \urcorner \mid \lambda x\,.\,e
\end{array}
$$

A.2. *Encoding Λ evaluation in Λ_{ml}*

Figures 14 and 15 give the call-by-name and call-by-value encodings of Λ in Λ_{ml}. In the call-by-name encoding, function arguments are passed unevaluated. This is reflected in the transformation on types, i.e., functions map computations to computations. In the call-by-value encoding, evaluation of function arguments is forced using the let con-

$$
\begin{array}{rcl}
\mathcal{E}_v\llbracket\cdot\rrbracket & : & \Lambda \rightarrow \Lambda_{ml} \\
\mathcal{E}_v\llbracket v\rrbracket & = & \mathsf{unit}\ \mathcal{E}_v\langle v\rangle \\
\mathcal{E}_v\llbracket f\rrbracket & = & f \\
\mathcal{E}_v\llbracket \mathsf{succ}\ e\rrbracket & = & \mathsf{let}\ y \Leftarrow \mathcal{E}_v\llbracket e\rrbracket\ \mathsf{in\ succ}\ y \\
\mathcal{E}_v\llbracket \mathsf{pred}\ e\rrbracket & = & \mathsf{let}\ y \Leftarrow \mathcal{E}_v\llbracket e\rrbracket\ \mathsf{in\ pred}\ y \\
\mathcal{E}_v\llbracket \mathsf{if0}\ e_0\ e_1\ e_2\rrbracket & = & \mathsf{let}\ y_0 \Leftarrow \mathcal{E}_v\llbracket e_0\rrbracket\ \mathsf{in\ if0}\ y_0\ \mathcal{E}_v\llbracket e_1\rrbracket\ \mathcal{E}_v\llbracket e_2\rrbracket \\
\mathcal{E}_v\llbracket e_0\ @\ e_1\rrbracket & = & \mathsf{let}\ y_0 \Leftarrow \mathcal{E}_v\llbracket e_0\rrbracket\ \mathsf{in\ let}\ y_1 \Leftarrow \mathcal{E}_v\llbracket e_1\rrbracket\ \mathsf{in}\ y_0\ @\ y_1 \\
\mathcal{E}_v\llbracket \mathsf{fix}\ f.\ e\rrbracket & = & \mathsf{fix}\ f.\ \mathcal{E}_v\llbracket e\rrbracket \\
\\
\mathcal{E}_v\langle\cdot\rangle & : & \mathit{Values}_v[\Lambda] \rightarrow \Lambda_{ml} \\
\mathcal{E}_v\langle\ulcorner n\urcorner\rangle & = & \ulcorner n\urcorner \\
\mathcal{E}_v\langle x\rangle & = & x \\
\mathcal{E}_v\langle\lambda x\ .\ e\rangle & = & \lambda x\ .\ \mathcal{E}_v\llbracket e\rrbracket \\
\\
\mathcal{E}_v\langle\mathsf{nat}\rangle & = & \mathsf{nat} \\
\mathcal{E}_v\langle\tau_1 \rightarrow \tau_2\rangle & = & \mathcal{E}_v\langle\tau_1\rangle \rightarrow \mathcal{E}_v\llbracket\tau_2\rrbracket \\
\mathcal{E}_v\llbracket\tau\rrbracket & = & \widetilde{\mathcal{E}_v\langle\tau\rangle} \\
\\
\mathcal{E}_v\llbracket\Gamma, x:\tau\rrbracket & = & \mathcal{E}_v\llbracket\Gamma\rrbracket, x:\mathcal{E}_v\langle\tau\rangle \\
\mathcal{E}_v\llbracket\Gamma, f:\tau\rrbracket & = & \mathcal{E}_v\llbracket\Gamma\rrbracket, f:\mathcal{E}_v\llbracket\tau\rrbracket
\end{array}
$$

Fig. 15. Call-by-value encoding $\mathcal{E}_v$ into Λ_{ml}

struct. This is reflected in the transformation on types, i.e., functions map values to computations.[†]

Appendix. References

Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, May 1994. DIKU Rapport 94/19.

Henk Barendregt. *The Lambda Calculus — Its Syntax and Semantics.* North-Holland, 1984.

P. N. Benton. *Strictness Analysis of Lazy Functional Programs.* PhD thesis, Computer Laboratory, University of Cambridge, Cambridge, England, May 1995.

[†] One can also give a call-by-value encoding where functions map computations to computations. Instead of forcing evaluation of function arguments in the application structure (as in $\mathcal{E}_v$), evaluation is forced immediately after an argument is received by a function (Hatcliff and Danvy 1994).

P. N. Benton, G. M. Bierman, and V. C. V de Paiva. Computational types from a logical perspective I. Technical report 365, Computer Laboratory, University of Cambridge, Cambridge, England, May 1995.

Hans-J. Boehm, editor. *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994. ACM Press.

Anders Bondorf. Improving binding times without explicit cps-conversion. In William Clinger, editor, *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. V, No. 1, pages 1–10, San Francisco, California, June 1992. ACM Press.

Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

Anders Bondorf and Dirk Dussart. Improving CPS-based partial evaluation: Writing cogen by hand. In Peter Sestoft and Harald Søndergaard, editors, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Technical Report 94/9, University of Melbourne, Australia, pages 1–10, Orlando, Florida, June 1994.

Charles Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In David A. Schmidt, editor, *Proceedings of the Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154, Copenhagen, Denmark, June 1993. ACM Press.

Charles Consel and Olivier Danvy. For a better support of static data flow. In Hughes (Hughes 1991), pages 496–519.

Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

Roy L. Crole and Andrew D. Gordon. Factoring an adequacy proof (preliminary report). In John T. O'Donnell and Kevin Hammond, editors, *Functional Programming, Glasgow 1993*, Workshops in Computing, pages 9–25, Ayr, Scotland, 1993. Springer-Verlag.

Roy L. Crole and Andrew M. Pitts. New foundations for fixpoint computations: FIX-hyperdoctrines and the FIX-logic. *Information and Computation*, 98:171–210, 1992.

Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.

Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

Olivier Danvy and John Hatcliff. CPS transformation after strictness analysis. *ACM Letters on Programming Languages and Systems*, 1(3):195–212, 1993.

Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. The essence of eta-expansion in partial evaluation. *LISP and Symbolic Computation*, 8(3):209–227, 1995. An earlier version appeared in the proceedings of the 1994 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation.

Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. Eta-expansion does The Trick. *ACM Transactions on Programming Languages and Systems*, 1996. To appear.

Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 258–283, St. Petersburg Beach, Florida, January 1996. ACM Press.

Dirk Dussart. PhD thesis, Katholieke Universiteit Leuven, Leuven, Belgium, 1997. Forthcoming.

Dirk Dussart and Peter Thiemann. Imperative functional specialization. Berichte des Wilhelm-Schickard-Instituts WSI-96-28, Universität Tübingen, July 1996.

R. Kent Dybvig, editor. *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, Pennsylvania, May 1996. ACM Press.

Andrzej Filinski. Representing monads. In Boehm (Boehm 1994), pages 446–457.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In David W. Wall, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 28, No 6, pages 237–247, Albuquerque, New Mexico, June 1993. ACM Press.

Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions for program specialization. In Hermenegildo and Swierstra (Hermenegildo and Swierstra 1995), pages 259–278.

Robert Glück and Andrei Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In Patrick Cousot, Moreno Falaschi, Gilberto Filè, and Antoine Rauzy, editors, *Proceedings of the Third International Workshop on Static Analysis WSA'93*, number 724 in Lecture Notes in Computer Science, pages 112–123, Padova, Italy, September 1993.

Carsten K. Gomard. A self-applicable partial evaluator for the lambda-calculus: Correctness and pragmatics. *ACM Transactions on Programming Languages and Systems*, 14(2):147–172, 1992.

Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, 1991.

Andrew D. Gordon. *Functional Programming and Input/Output*. PhD thesis, Computer Laboratory, University of Cambridge, Cambridge, England, February 1993. Technical Report No. 285.

Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.

John Hannan and Dale Miller. Deriving mixed evaluation from standard evaluation for a simple functional language. In J. L. A. van de Snepscheut, editor, *Proceedings of a Conference on Mathematics of Program Construction*, number 375 in Lecture Notes in Computer Science, pages 239–255, Groningen, The Netherlands, 1989.

John Hatcliff. *The Structure of Continuation-Passing Styles*. PhD thesis, Department of Computing and Information Sciences, Kansas State University, Manhattan, Kansas, June 1994.

John Hatcliff. Mechanically verifying the correctness of an offline partial evaluator. In Hermenegildo and Swierstra (Hermenegildo and Swierstra 1995), pages 279–298.

John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Boehm (Boehm 1994), pages 458–471.

John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation (extended version). Technical Report BRICS RS-96-34, Computer Science Department, Aarhus University, Aarhus, Denmark, 1996.

John Hatcliff and Robert Glück. An operational theory of self-applicable on-line program specialization. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, pages 161–182, Dagstuhl, Germany, February 1996.

Manuel Hermenegildo and S. Doaitse Swierstra, editors. *Seventh International Symposium on Programming Language Implementation and Logic Programming*, number 982 in Lecture Notes in Computer Science, Utrecht, The Netherlands, September 1995.

Carsten K. Holst and Carsten K. Gomard. Partial evaluation is fuller laziness. In Paul Hudak and Neil D. Jones, editors, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN Notices, Vol. 26, No 9, pages 223–233, New Haven, Connecticut, June 1991. ACM Press.

John Hughes, editor. *Proceedings of the Fifth ACM Conference on Functional Programming and Computer Architecture*, number 523 in Lecture Notes in Computer Science, Cambridge, Massachusetts, August 1991. Springer-Verlag.

Neil D. Jones, editor. *Special issue on Partial Evaluation*, Journal of Functional Programming, Vol. 3, Part 3. Cambridge University Press, July 1993.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International Series in Computer Science. Prentice-Hall, 1993.

Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: A self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.

John Launchbury. A strongly-typed self-applicable partial evaluator. In Hughes (Hughes 1991), pages 145–164.

Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. In Carolyn L. Talcott, editor, *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. VII, No. 3, Orlando, Florida, June 1994. ACM Press.

Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. Technical Report CS-95-178, Computer Science Department, Brandeis University, Waltham, Massachusetts, January 1995. An earlier version appeared in the proceedings of the 1994 ACM Conference on Lisp and Functional Programming.

Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In Peter Lee, editor, *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 333–343, San Francisco, California, January 1995. ACM Press.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

Kristian Nielsen. Master's thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1997. Forthcoming.

Kristian Nielsen and Morten Heine Sørensen. Call-by-name CPS-translation as a binding-time improvement. In Alan Mycroft, editor, *Static Analysis*, number 983 in Lecture Notes in Computer Science, pages 296–313, Glasgow, Scotland, September 1995. Springer-Verlag.

Flemming Nielson. A denotational framework for data flow analysis. *Acta Informatica*, 18:265–287, 1982.

Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications, a formal introduction*. Wiley Professional Computing. John Wiley and Sons, 1992.

Jens Palsberg. Correctness of binding-time analysis. In Jones (Jones 1993), pages 347–363.

Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

Dag Prawitz. *Natural Deduction*. Almquist and Wiksell, Uppsala, 1965.

Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6(3/4):289–360, December 1993.

Amr Sabry and Matthias Felleisen. Is continuation-passing useful for data flow analysis? In Vivek Sarkar, editor, *Proceedings of the ACM SIGPLAN'94 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 29, No 6, pages 1–12, Orlando, Florida, June 1994. ACM Press.

Amr Sabry and Philip Wadler. Compiling with reflections. In Dybvig (Dybvig 1996), pages 13–24.

David Sands. Proving the correctness of recursion-based automatic program transformations. In Peter Mosses, Mogens Nielsen, and Michael Schwartzbach, editors, *Proceedings of TAPSOFT '95*, number 915 in Lecture Notes in Computer Science, pages 681–695, Aarhus, Denmark, May 1995.

Morten Heine Sørensen, Robert Glück, and Neil Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In Donald Sannella, editor, *Proceedings of the Fifth European Symposium on Programming*, number 788 in Lecture Notes in Computer Science, pages 485–500, Edinburgh, Scotland, April 1994.

Guy L. Steele Jr. Building interpreters by composing monads. In Boehm (Boehm 1994), pages 472–492.

Carolyn Talcott. A theory for program and data type specification. *Theoretical Computer Science*, 104(1):129–159, 1992.

Peter Thiemann. Cogen in six lines. In Dybvig (Dybvig 1996), pages 180–189.

Philip Wadler. The essence of functional programming (tutorial). In Andrew W. Appel, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 1992. ACM Press.

Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, December 1992.

Mitchell Wand. Specifying the correctness of binding-time analysis. In Jones (Jones 1993), pages 365–387.