A Generic Framework for Specialization (Abridged Version)

Peter Thiemann*

Universität Tübingen
thiemann@informatik.uni-tuebingen.de

Abstract. We present a generic framework for specifying and implementing offline partial evaluators. The framework provides the infrastructure for specializing higher-order programs with computational effects specified through a monad. It performs sound specialization for all monadic instances and is evaluation-order independent. It subsumes most previously published partial evaluators for higher-order functional programming languages in the sense that they are instances of the generic framework with respect to a particular monad.

Keywords: higher-order programming, program transformation, partial evaluation, computational effects

1 Introduction

A *partial evaluator* [10,23] specializes a program with respect to a known part of its input. The resulting specialized program takes the rest of the input and delivers the same result as the original program applied to the whole input. The specialized program usually runs faster than the original one.

One particular flavor of partial evaluation is *offline* partial evaluation. In its first stage, a *binding-time analysis* annotates all phrases of a program that only depend on the known input as executable at specialization time. The execution of the remaining phrases is deferred to the run time of the specialized program. In the second stage, a *static reducer* interprets the annotated program. It evaluates all phrases annotated as executable and generates code for the remaining phrases.

The goal of this work is to present a generic framework to specify and implement the static reducer. The framework unifies the existing specifications of static reducers and it provides a sound basis to implement reducers that execute and generate code with computational effects. It goes beyond existing specializers in that it allows for experimentation with various effects in a modular way.

To achieve this modularity, the framework is parameterized over a monad. The choice of a monad fixes a particular computational effect. Monads have been used in the context of programming languages to structure denotational

* Author's present address: Department of Computer Science, University of Nottingham, University Park, Nottingham NG7 2RD, England

semantics [30], to structure functional programs [39,40], and to equip pure functional languages with side-effecting operations like I/O and mutable state [24,32]. Closely related to structuring denotational semantics is the construction of modular interpreters [18,28]. There are also recent theoretical approaches to formalize partial evaluation using monads [21,26].

1.1 Four Ways to Static Reduction

In the past, four different approaches have been used to implement static reducers dealing with a particular computational effect. All of them rest on denotational specifications of specialization, or—from a programmer's point of view—on viewing the static reducer as an interpreter for an annotated language. The implementation languages of these interpreters (the metalanguages of the specifications) range from applied lambda calculus to ML with control operators. By factoring these specifications over an annotated variant of Moggi's computational metalanguage [21,30], we demonstrate that all of them are composed from the same set of building blocks, the sole difference being the staging of computation at specialization time. Here is the set of building blocks:

eval_v	an evaluation function for a pure call-by-value lambda calculus;
$\mathcal{B}$	a binding-time analysis that maps terms to annotated terms;
$\mathcal{S}_v$	a specializer for applied lambda calculus written in lambda calculus (e.g., Lambdamix [19]);
$\mathcal{M}$	a *monadic expansion* translation that maps the computational metalanguage to applied lambda calculus by expanding the monadic operators to lambda terms according to the definition of the monad;
$\mathcal{E}_v$	a call-by-value *explication* that translates from the (annotated) source language to the (annotated) metalanguage, encoding a call-by-value evaluation strategy.

As an example for the different approaches consider a call-by-value language $\lambda_!$ with some computational effect and specialize a program p to r with respect to known data s. We assume access to the program text of all functions mentioned above: $\lceil \mathcal{S}_v \rceil$ is the lambda term denoting the function $\mathcal{S}_v = \mathrm{eval}_v\ \lceil \mathcal{S}_v \rceil$, $\lceil p\ s \rceil$ is the textual application of p to s, and so on.

Transform Source Program to Expanded Monadic Style Applying $\mathcal{M} \circ \mathcal{E}_v$ to $\lceil p\ s \rceil$ yields an effect-free lambda term. This term can now be analyzed and statically reduced with a specializer for the lambda calculus [8,19].

$$\mathcal{S}_v\ (\mathcal{B}\ (\mathcal{M}\ (\mathcal{E}_v(\lceil p\ s \rceil)))) = \mathcal{M}\ (\mathcal{E}_v(\lceil r \rceil)) \tag{1}$$

This approach is viable [9,31,37], but it suffers from a number of drawbacks.

- Monadic expansion typically introduces many new abstractions. This increase in program size slows down the analysis $\mathcal{B}$ and the static reduction.

- The expanded term can be hard to read for the user of the partial evaluator. It provides no useful feedback from the annotated term on how to change the source program to achieve better specialization.
- The straightforward expansion often does not lead to satisfactory results from the binding-time analysis. For example, sophisticated state-passing translations have been designed to obtain better results [31,37].
- The specialized program is also in expanded monadic style (for example, in continuation-passing style [9]), which requires an inverse translation (for example, a direct style translation [13]) to obtain readable results.

Specializer in Expanded Monadic Style At the cost of moving to a more sophisticated binding-time analysis $\mathcal{B}_!$, which takes computational effects into account (e.g., [37]), equation (1) can be rewritten to

$$\begin{array}{lll} & \mathcal{S}_v\ (\mathcal{M}\ (\mathcal{E}_v(\mathcal{B}_!\lceil p\ s\rceil))) & = \mathcal{M}\ (\mathcal{E}_v\lceil r\rceil) \\ \text{hence} & (\mathcal{S}_v \circ \mathcal{M} \circ \mathcal{E}_v)\ (\mathcal{B}_!\lceil p\ s\rceil) & = (\mathcal{M} \circ \mathcal{E}_v)\ \lceil r\rceil \\ \text{hence} & (\mathcal{E}_v^{-1} \circ \mathcal{M}^{-1} \circ \mathcal{S}_v \circ \mathcal{M} \circ \mathcal{E}_v)\ (\mathcal{B}_!\lceil p\ s\rceil) & = \lceil r\rceil \end{array}$$

Changing the staging by symbolically composing the specializer with the monadic expansion and the explication and their inverses yields

$$\text{eval}_v\ \lceil \mathcal{E}_v^{-1} \circ \mathcal{M}^{-1} \circ \mathcal{S}_v \circ \mathcal{M} \circ \mathcal{E}_v\rceil\ \mathcal{B}_!\lceil p\ s\rceil = \lceil r\rceil \qquad (2)$$

There are a number of advantages in return for this complication.

- The source program does not undergo any translation.
- The binding-time analysis applies directly to the source program p and can transmit useful feedback to the user of the system.
- The specialized program is built from pieces of the original source program. The inverse translation is hard-wired into the specializer.

Examples of this approach are Bondorf's specializer in extended continuation-passing style [4] and a specializer for call-by-value lambda calculus with first-class references in extended continuation-passing store-passing style [17].

Specializer in Direct Style with Monadic Operators Here we depart from writing the specializer in a pure language and use a meta-interpreter $\text{eval}_! = (\text{eval}_v \circ \mathcal{M} \circ \mathcal{E}_v)$ for $\lambda_!$ with the monad in question built in. On top of that we write a direct style specializer $\mathcal{S}_!$ in $\lambda_!$ using the built-in monadic operations. For example, if $\mathcal{S}_!$ was written in ML it could make use of exceptions, state, and control operations.

This approach has the same advantages as the previous approach. Additionally, it is usually more efficient since $\text{eval}_!$ can employ machine-level implementations of the monadic operations. Now we can reason as follows:

$$\begin{aligned} \lceil r\rceil &= \text{eval}_!\ \lceil \mathcal{S}_!\rceil\ (\mathcal{B}_!\lceil p\ s\rceil) \\ &= (\text{eval}_v \circ \mathcal{M} \circ \mathcal{E}_v)\ \lceil \mathcal{S}_!\rceil\ (\mathcal{B}_!\lceil p\ s\rceil) \\ &= \text{eval}_v\ (\mathcal{M}(\mathcal{E}_v\lceil \mathcal{S}_!\rceil))\ (\mathcal{B}_!\lceil p\ s\rceil) \end{aligned}$$

Lawall and Danvy [25] have followed exactly this path (for the continuation monad) to construct an efficient implementation of Bondorf's specializer in direct style with control operators. Their work exploits the built-in continuation monad of Scheme and ML.

Since $\mathcal{S}_!$ simply interprets unannotated pure terms, it follows that

$$\mathcal{M}(\mathcal{E}_v\lceil\mathcal{S}_!\rceil) = \lceil\mathcal{S}_! \circ \mathcal{M} \circ \mathcal{E}_v\rceil$$

This generalizes a result of Lawall and Danvy [25], namely $\lceil\mathcal{S}_c\rceil = \mathcal{C}_v\lceil\mathcal{S}_!\rceil$ where

$\mathcal{C}_v$ is a translation to extended continuation-passing style, encoding call-by-value evaluation; it holds [20] that $\mathcal{C}_v = \mathcal{M}_c \circ \mathcal{E}_v$ (where $\mathcal{M}_c$ is the monadic expansion for the continuation monad, see Sec. 5);
$\mathcal{S}_c$ is a continuation-based specializer in CPS; it holds that $\mathcal{S}_c = \mathcal{S}_v \circ \mathcal{C}_v$.

Define a Specializer for the Metalanguage Hatcliff and Danvy [21] translate p to the metalanguage via $\mathcal{E}_v$ and then define the binding-time analysis $\mathcal{B}_{ML}$ and the specializer $\mathcal{S}_{ML}$ for the metalanguage.

$$\mathcal{S}_{ML}\ (\mathcal{B}_{ML}[\![\mathcal{E}_v(\lceil p\ s\rceil)]\!]) = \mathcal{E}_v(\lceil r\rceil)$$

The specialized program is also written in the metalanguage and may have to be translated back into $\lambda_!$.

1.2 The Design Space

Figure 1 shows the design space of specialization for languages with computational effects. Again, $\lambda_!$ is an impure lambda calculus with some "built-in" effects, λ_{ML} is an enrichment of Moggi's computational metalanguage, and λ is a pure (but applied) lambda calculus. The underlined variants are the respective annotated versions of the calculi. They are connected via binding-time analyses $\mathcal{B}$ for the respective calculi. The annotated programs are mapped to their specialized versions by the respective specializer $\mathcal{S}$. The diagram should convey the idea that—ideally—the order of the transformations should not matter for the results. Paths in the diagram correspond to design choices in constructing a specializer, as exemplified before. Clearly, if the path to the specialized program does not end in the lower left of the diagram, we need inverse transformations for $\mathcal{E}$ and (possibly) $\mathcal{M}$ to get a specialized program in the source language.

The diagram suggests that a specializer has to map the annotated version of a calculus to its standard version. As a counterexample, consider the specializer $\mathcal{M}^{-1} \circ \mathcal{S}_!$ that maps $\underline{\lambda}_!$ to λ_{ML}.

1.3 Our Approach

We distinguish three languages, the annotated source language $\underline{\lambda}_!$, the annotated computational metalanguage $\underline{\lambda}_{ML}$, and an *implementation language* λimpl, which

$$\begin{tikzcd}
\boxed{\lambda_!} \arrow[r, "\mathcal{E}"] \arrow[d, "\mathcal{B}_!"] & \boxed{\lambda_{ML}} \arrow[r, "\mathcal{M}"] \arrow[d, "\mathcal{B}_{ML}"] & \boxed{\lambda} \arrow[d, "\mathcal{B}"] \\
\boxed{\underline{\lambda}_!} \arrow[r, "\mathcal{E}"] \arrow[d, "\mathcal{S}_!"] & \boxed{\underline{\lambda}_{ML}} \arrow[r, "\mathcal{M}"] \arrow[d, "\mathcal{S}_{ML}"] & \boxed{\underline{\lambda}} \arrow[d, "\mathcal{S}"] \\
\boxed{\lambda_!} \arrow[r, "\mathcal{E}"] & \boxed{\lambda_{ML}} \arrow[r, "\mathcal{M}"] & \boxed{\lambda}
\end{tikzcd}$$

Fig. 1. Generic Framework

is a functional programming language equipped with a particular monad T . All of them are defined in Sec. 2. In Sec. 3, we define a monadic semantics for the annotated source language in terms of an explication translation from $\underline{\lambda}_!$ to $\underline{\lambda}_{ML}$. The actual specializer maps $\underline{\lambda}_{ML}$ programs to $\lambda_!$ programs. This has two advantages: the binding-time analysis and the explication of the evaluation order are left to the frontend and an inverse of the explication translation is not required. The first stage of specialization, $\mathcal{G}$, maps $\underline{\lambda}_{ML}$ programs to λimpl programs. $\mathcal{G}$ is generalized from a continuation interpretation for $\underline{\lambda}_{ML}$ in Sec. 4. Evaluation of the λimpl program yields the specialized program in $\lambda_!$. Subsequently, we instantiate T with various monads and discuss the outcome: the continuation monad yields continuation-based partial evaluation (Sec. 5), the identity monad yields Gomard and Jones's specializer for the lambda calculus (Sec. 6), a combination of the continuation monad and the store monad yields a specializer for core ML with references (Sec. 7), a combination of the continuation monad and the exception monad yields a specializer that can process exceptions at specialization time (Sec. 8). For each choice of monad T we give a specification $\mathcal{G}_{\mathrm{T}}$ of the monadic operators in terms of λ. The composition $\mathcal{G}_{\mathrm{T}} \circ \mathcal{G}$ plays the role of $\mathcal{M}$ in the diagram. As outlined in 1.1 above, any implementation of the operators will do, as long as it obeys the specification.

2 Notation

2.1 Annotated Lambda Calculus

The source language is a simply typed annotated lambda calculus $\underline{\lambda}$. It will be extended later to $\underline{\lambda}_!$ to demonstrate the treatment of monadic effects like state and exceptions. It is straightforward to extend both with the usual programming constructs. The typing rules are standard.

$$\begin{array}{l} \text{terms } E ::= x \mid \lambda x.E \mid E@E \mid \underline{\lambda} x.E \mid E\underline{@}E \\ \text{types } \tau \ ::= \iota \mid \tau \rightarrow \tau \mid \underline{\iota} \mid \tau \underline{\rightarrow} \tau \end{array}$$

The type ι is the type of integers, and $\tau \rightarrow \tau'$ is the type of functions that map τ to τ'. In the implementation, the underlined (dynamic) types are subsumed in the type Code. Beta reduction of static terms is the only rule of computation.

We use standard notational conventions: application associates to the left, the scope of a λ goes as far to the right as possible, and we can merge lambda abstractions as in $\lambda xy.E$. As usual, $\Rightarrow$ is single step reduction, $\Rightarrow\!\!\!\!\Rightarrow$ is the reflexive transitive closure of $\Rightarrow$, and $=$ is the reflexive, transitive, and symmetric closure of $\Rightarrow$.

2.2 Annotated Computational Metalanguage

The computational metalanguage [30] is an extended lambda calculus that makes the introduction and composition of computations explicit. Here is the syntax of its annotated version [21]:

$$\begin{array}{ll} \text{terms } M ::= & x \mid \lambda x.M \mid M@M \mid \text{unit}(M) \mid \text{let } x \Leftarrow M \text{ in } M \\ & \mid \underline{\lambda} x.M \mid M\underline{@}M \mid \underline{\text{unit}}(M) \mid \underline{\text{let}}\ x \Leftarrow M \ \underline{\text{in}}\ M \\ \text{types } \tau \ ::= & \iota \mid \tau \rightarrow \tau \mid \mathrm{T}\ \tau \mid \underline{\iota} \mid \tau \underline{\rightarrow} \tau \mid \underline{\mathrm{T}}\ \tau \end{array}$$

Intuitively, unit(M) denotes a trivial computation that returns the value of M. The *monadic let* let $x \Leftarrow M_1$ in M_2 expresses the sequential composition of computations: first, M_1 is performed and then M_2 with x bound to the value returned by M_1. We augment beta reduction and the monadic reduction rules

$$\text{let } x \Leftarrow (\text{let } y \Leftarrow M_1 \text{ in } M_2) \text{ in } M_3 \rightarrow \text{let } y \Leftarrow M_1 \text{ in let } x \Leftarrow M_2 \text{ in } M_3 \quad (3)$$

$$\text{let } x \Leftarrow M \text{ in unit}(x) \rightarrow M \quad (4)$$

$$\text{let } x \Leftarrow \text{unit}(M_1) \text{ in } M_2 \rightarrow M_2[x := M_1] \quad (5)$$

by rules that reorganize underlined let expressions, too.

$$\text{let } x \Leftarrow (\underline{\text{let}}\ y \Leftarrow M_1 \ \underline{\text{in}}\ M_2) \text{ in } M_3 \rightarrow \underline{\text{let}}\ y \Leftarrow M_1 \ \underline{\text{in}} \text{ let } x \Leftarrow M_2 \text{ in } M_3 \quad (6)$$

$$\underline{\text{let}}\ x \Leftarrow (\text{let } y \Leftarrow M_1 \text{ in } M_2) \ \underline{\text{in}}\ M_3 \rightarrow \text{let } y \Leftarrow M_1 \text{ in } \underline{\text{let}}\ x \Leftarrow M_2 \ \underline{\text{in}}\ M_3 \quad (7)$$

$$\underline{\text{let}}\ x \Leftarrow (\underline{\text{let}}\ y \Leftarrow M_1 \ \underline{\text{in}}\ M_2) \ \underline{\text{in}}\ M_3 \rightarrow \underline{\text{let}}\ y \Leftarrow M_1 \ \underline{\text{in}}\ \underline{\text{let}}\ x \Leftarrow M_2 \ \underline{\text{in}}\ M_3 \quad (8)$$

2.3 Implementation Language

The implementation language λimpl is a lambda-calculus extended with the monadic constructs and some special operators. The special operators include the binary syntax constructors $\hat{\lambda}(,)$ and $\hat{@}(,)$. Furthermore, there are operators specific to the currently used monad (see below for examples). The implementation language is no longer a true annotated language, since $\hat{\lambda}(,)$ and $\hat{@}(,)$ are merely constructors for the datatype Code. The implementation language is purposefully close to existing functional programming languages so that its programs are easily transcribed.

3 Explication

The explication translation performs the first part of the work. It maps the (annotated) source language $\underline{\lambda}_!$ into the (annotated) metalanguage $\underline{\lambda}_{ML}$ and makes a particular evaluation order explicit. We only consider $\mathcal{E}_v()$ which fixes left-to-right call-by-value evaluation.

$$\begin{array}{ll}
\mathcal{E}_v(x) & \equiv \mathrm{unit}(x) \\
\mathcal{E}_v(\lambda x.E) & \equiv \mathrm{unit}(\lambda x.\mathcal{E}_v(E)) \\
\mathcal{E}_v(E_1@E_2) & \equiv \mathrm{let}\ x_1 \Leftarrow \mathcal{E}_v(E_1)\ \mathrm{in}\ \mathrm{let}\ x_2 \Leftarrow \mathcal{E}_v(E_2)\ \mathrm{in}\ x_1@x_2 \\
\mathcal{E}_v(\underline{\lambda} x.E) & \equiv \underline{\mathrm{unit}}(\underline{\lambda} x.\mathcal{E}_v(E)) \\
\mathcal{E}_v(E_1\underline{@}E_2) & \equiv \underline{\mathrm{let}}\ x_1 \Leftarrow \mathcal{E}_v(E_1)\ \underline{\mathrm{in}}\ \underline{\mathrm{let}}\ x_2 \Leftarrow \mathcal{E}_v(E_2)\ \underline{\mathrm{in}}\ x_1\underline{@}x_2
\end{array}$$

The translation of the underlined constructs mirrors the translation of the static constructs exactly, which in turn is standard [20]. This will always be the case for the explication translation. It is easy to show that the translation preserves typing.

Lemma 1. *Suppose $\Gamma \vdash E : \tau$. Then $\mathcal{M}_v[\![\Gamma]\!] \vdash \mathcal{E}_v(E) : \mathcal{M}_{cv}[\![\tau]\!]$ where*

$$\begin{array}{lll}
\mathcal{M}_{cv}[\![\tau]\!] = \mathrm{T}\ \mathcal{M}_v[\![\tau]\!] & & \\
\mathcal{M}_v[\![\iota]\!] = \iota & \mathcal{M}_v[\![\tau_2 \rightarrow \tau_1]\!] = \mathcal{M}_v[\![\tau_2]\!] \rightarrow \mathcal{M}_{cv}[\![\tau_1]\!] & \mathcal{M}_v[\![\mathrm{Code}]\!] = \mathrm{Code} \\
\mathcal{M}_v[\![\{\}]\!] = \{\} & \mathcal{M}_v[\![\Gamma\{x : \tau\}]\!] = \mathcal{M}_v[\![\Gamma]\!]\{x : \mathcal{M}_v[\![\tau]\!]\} &
\end{array}$$

This approach is similar to that of Hatcliff and Danvy [21]. They translate the source language to the metalanguage in the very beginning and perform the binding-time analysis on terms of the metalanguage. We can accommodate this setup, but we also support a translation from the annotated source language (after binding-time analysis) to the annotated metalanguage. Both metalanguages have in common the existence of reductions involving the underlined constructs. The language considered by Hatcliff and Danvy excludes rules (7) and (8) [21, fig. 10]. This is merely a different design choice as explained by Lawall and Thiemann [26]. The crucial point is the following theorem [21].

Theorem 1. *$\underline{\lambda}_{ML}$ reduction preserves operational equivalence.*

4 Semantics for $\underline{\lambda}_{ML}$

The language $\underline{\lambda}_{ML}$ contains non-standard reductions, namely the reorganizing rules for let expressions. Hence we develop a CPS translation that maps $\underline{\lambda}_{ML}$ to $\underline{\lambda}$ in such a way that reduction in $\underline{\lambda}_{ML}$ is simulated by reduction in λimpl. Figure 2 defines the translation using $\widehat{\mathrm{let}}(x, E_1, E_2)$ as syntactic sugar for $\hat{@}((\hat{\lambda}(x, E_2)), E_1)$.

Lemma 2. *Suppose $M_1 \Rrightarrow_{ML} M_2$ then $\mathcal{M}_c[\![M_1]\!] =_{\lambda impl} \mathcal{M}_c[\![M_2]\!]$.*

Recall that we promised a generic implementation scheme parameterized over a monad. So far, we have only produced one implementation for a particular instance, the continuation monad. Let us now abstract from this instance

$$
\begin{array}{ll}
\mathcal{M}_c\llbracket x\rrbracket & \equiv x \\
\mathcal{M}_c\llbracket \lambda x.M\rrbracket & \equiv \lambda x.\mathcal{M}_c\llbracket M\rrbracket \\
\mathcal{M}_c\llbracket M_1@M_2\rrbracket & \equiv \lambda k.\mathcal{M}_c\llbracket M_1\rrbracket@\mathcal{M}_c\llbracket M_2\rrbracket@k \\
\mathcal{M}_c\llbracket \mathrm{unit}(M)\rrbracket & \equiv \lambda k.k@\mathcal{M}_c\llbracket M\rrbracket \\
\mathcal{M}_c\llbracket \mathrm{let}\ x \Leftarrow M_1\ \mathrm{in}\ M_2\rrbracket & \equiv \lambda k.\mathcal{M}_c\llbracket M_1\rrbracket@\lambda x.\mathcal{M}_c\llbracket M_2\rrbracket@k \\
\mathcal{M}_c\llbracket \underline{\lambda} x.M\rrbracket & \equiv \hat{\lambda}(x, \mathcal{M}_c\llbracket M\rrbracket@\lambda z.z) \\
\mathcal{M}_c\llbracket M_1\underline{@}M_2\rrbracket & \equiv \lambda k.k@(\hat{@}(\mathcal{M}_c\llbracket M_1\rrbracket, (\mathcal{M}_c\llbracket M_2\rrbracket@\lambda z.z)))\ \text{if}\ M_2 : \underline{\mathrm{T}}\ \tau \\
 & \equiv \lambda k.k@(\hat{@}(\mathcal{M}_c\llbracket M_1\rrbracket, \mathcal{M}_c\llbracket M_2\rrbracket)) \\
\mathcal{M}_c\llbracket \underline{\mathrm{unit}}(M)\rrbracket & \equiv \lambda k.k@\mathcal{M}_c\llbracket M\rrbracket \\
\mathcal{M}_c\llbracket \underline{\mathrm{let}}\ x \Leftarrow M_1\ \underline{\mathrm{in}}\ M_2\rrbracket & \equiv \lambda k.\mathcal{M}_c\llbracket M_1\rrbracket@\lambda z.\widehat{\mathrm{let}}(x, z, \mathcal{M}_c\llbracket M_2\rrbracket@k)
\end{array}
$$

Fig. 2. Annotated continuation introduction

$$
\begin{array}{ll}
\mathcal{G}\llbracket x\rrbracket & \equiv x \\
\mathcal{G}\llbracket \lambda x.M\rrbracket & \equiv \lambda x.\mathcal{G}\llbracket M\rrbracket \\
\mathcal{G}\llbracket M_1@M_2\rrbracket & \equiv \mathrm{apply}\ \mathcal{G}\llbracket M_1\rrbracket\ \mathcal{G}\llbracket M_2\rrbracket \\
\mathcal{G}\llbracket \mathrm{unit}(M)\rrbracket & \equiv \mathrm{unit}(\mathcal{G}\llbracket M\rrbracket) \\
\mathcal{G}\llbracket \mathrm{let}\ x \Leftarrow M_1\ \mathrm{in}\ M_2\rrbracket & \equiv \mathrm{let}\ x \Leftarrow \mathcal{G}\llbracket M_1\rrbracket\ \mathrm{in}\ \mathcal{G}\llbracket M_2\rrbracket \\
\mathcal{G}\llbracket \underline{\lambda} x.M\rrbracket & \equiv \hat{\lambda}(x, \sharp_{\mathrm{Code}}(\mathcal{G}\llbracket M\rrbracket)) \\
\mathcal{G}\llbracket M_1\underline{@}M_2\rrbracket & \equiv \mathrm{unit}(\hat{@}(\mathcal{G}\llbracket M_1\rrbracket, \sharp_{\mathrm{Code}}(\mathcal{G}\llbracket M_2\rrbracket)))\ \text{if}\ M_2 : \underline{\mathrm{T}}\ \tau \\
 & \equiv \mathrm{unit}(\hat{@}(\mathcal{G}\llbracket M_1\rrbracket, \mathcal{G}\llbracket M_2\rrbracket))\ \text{otherwise} \\
\mathcal{G}\llbracket \underline{\mathrm{unit}}(M)\rrbracket & \equiv \mathrm{unit}(\mathcal{G}\llbracket M\rrbracket) \\
\mathcal{G}\llbracket \underline{\mathrm{let}}\ x \Leftarrow M_1\ \underline{\mathrm{in}}\ M_2\rrbracket & \equiv \mathrm{shift}_{\mathrm{Code}}\ w.\,\mathrm{let}\ z \Leftarrow \mathcal{G}\llbracket M_1\rrbracket\ \mathrm{in} \\
 & \qquad \mathrm{unit}(\widehat{\mathrm{let}}(x, z, \sharp_{\mathrm{Code}}(w@\mathcal{G}\llbracket M_2\rrbracket)))
\end{array}
$$

Fig. 3. Translation to the implementation language

by rewriting—in the implementation language—the right sides of $\mathcal{M}_c$ such that we obtain the original definition of $\mathcal{M}_c$ by monadic expansion, i.e., CPS transformation.

Figure 3 defines the translation. It might seem like nothing has happened in this transformation. However, we have taken an important conceptual step. We have gotten rid of the annotated language with non-standard reductions in favor of a standard functional language with monadic operators. In other words, we have an implementation.

There are two non-standard constructs in the translated terms. $\sharp_{\mathrm{Code}}(M)$ denotes an effect delimiter. It runs the computation M (which may involve effects), discards all effects, and returns the result, provided that M terminates. $\mathrm{shift}_{\mathrm{Code}}\ x.M$ grabs the context of the computation up to the next enclosing $\sharp_{\mathrm{Code}}(M')$, discards it, and binds it as a function to x. The standard implementation of these operators in terms of continuations is given in the next section 5.

These operators are restricted so that $\text{shift}_{\text{Code}}\ x.M$ only abstracts contexts that return Code. Otherwise, type soundness could not be guaranteed.

5 Continuation-Based Specialization

We obtain sound continuation-based specialization [26] from the generic specification $\mathcal{G}$ by substituting the continuation monad $\mathrm{T}\ \tau = (\tau \to \text{Code}) \to \text{Code}$ in the implementation language.

The monadic expansion translation boils down to defining the operators $\text{apply}\ M\ M$, $\text{unit}(M)$, $\text{let}\ x \Leftarrow M\ \text{in}\ M$, $\sharp_{\text{Code}}(M)$, and $\text{shift}_{\text{Code}}\ x.M$.

$$\begin{array}{ll}
\mathcal{G}_c[\![x]\!] & \equiv x \\
\mathcal{G}_c[\![\lambda x.M]\!] & \equiv \lambda x.\mathcal{G}_c[\![M]\!] \\
\mathcal{G}_c[\![\text{apply}\ M_1\ M_2]\!] & \equiv \lambda k.\mathcal{G}_c[\![M_1]\!]@\mathcal{G}_c[\![M_2]\!]@k \\
\mathcal{G}_c[\![\hat{\lambda}(x,M)]\!] & \equiv \hat{\lambda}(x,\mathcal{G}_c[\![M]\!]) \\
\mathcal{G}_c[\![\hat{@}(M_1,M_2)]\!] & \equiv \hat{@}(\mathcal{G}_c[\![M_1]\!],\mathcal{G}_c[\![M_2]\!]) \\
\mathcal{G}_c[\![\text{unit}(M)]\!] & \equiv \lambda k.k@\mathcal{G}_c[\![M]\!] \\
\mathcal{G}_c[\![\text{let}\ x \Leftarrow M_1\ \text{in}\ M_2]\!] & \equiv \lambda k.\mathcal{G}_c[\![M_1]\!]@\lambda x.\mathcal{G}_c[\![M_2]\!]@k \\
\mathcal{G}_c[\![\sharp_{\text{Code}}(M)]\!] & \equiv \mathcal{G}_c[\![M]\!]@\lambda x.x \\
\mathcal{G}_c[\![\text{shift}_{\text{Code}}\ x.M]\!] & \equiv \lambda k.(\lambda x.\mathcal{G}_c[\![M]\!]@\lambda z.z)@\lambda y.\lambda k'.k'@(k@y)
\end{array}$$

Here is the connection to $\mathcal{M}_c$ with $\circ$ denoting composition.

Lemma 3. $\mathcal{G}_c \circ \mathcal{G} \Rightarrow\!\!\!\!\Rightarrow_\beta \mathcal{M}_c$.

For the specializer $\mathcal{S}_c$ of Lawall and Thiemann [26] we find:

Lemma 4. $(\mathcal{G}_c \circ \mathcal{G} \circ \mathcal{E}_v) \Rightarrow\!\!\!\!\Rightarrow_\beta \mathcal{S}_c$.

There are specifications of continuation-based specialization in direct style [25] that employ the control operators shift and reset [14]. In contrast to reset, $\sharp_{\text{Code}}()$ is a general effect delimiter that runs an encapsulated computation, extracts its results, and hides all its effects. The difference is visible in the computation type of "reset(M)".

6 Specialization for the Lambda Calculus

Gomard's specializer Lambdamix [19] is targeted towards an applied lambda calculus. It results from setting $\mathrm{T}\ \tau = \tau$, the identity monad.

$$\begin{array}{llll}
\mathcal{G}_i[\![x]\!] & \equiv x & \mathcal{G}_i[\![\text{unit}(M)]\!] & \equiv \mathcal{G}_i[\![M]\!] \\
\mathcal{G}_i[\![\lambda x.M]\!] & \equiv \lambda x.\mathcal{G}_i[\![M]\!] & \mathcal{G}_i[\![\text{let}\ x \Leftarrow M_1\ \text{in}\ M_2]\!] & \equiv (\lambda x.\mathcal{G}_i[\![M_2]\!])@\mathcal{G}_i[\![M_1]\!] \\
\mathcal{G}_i[\![\text{apply}\ M_1\ M_2]\!] & \equiv \mathcal{G}_i[\![M_1]\!]@\mathcal{G}_i[\![M_2]\!] & \mathcal{G}_i[\![\sharp_{\text{Code}}(M)]\!] & \equiv \mathcal{G}_i[\![M]\!] \\
\mathcal{G}_i[\![\hat{\lambda}(x,M)]\!] & \equiv \hat{\lambda}(x,\mathcal{G}_i[\![M]\!]) & \mathcal{G}_i[\![\text{shift}_{\text{Code}}\ x.M]\!] & \equiv (\lambda x.\mathcal{G}_i[\![M]\!])@(\lambda z.z) \\
\mathcal{G}_i[\![\hat{@}(M_1,M_2)]\!] & \equiv \hat{@}(\mathcal{G}_i[\![M_1]\!],\mathcal{G}_i[\![M_2]\!]) & &
\end{array}$$

Calling Lambdamix $\mathcal{S}_i$ we have the following connection (see also [38]):

Lemma 5. $(\mathcal{G}_\iota \circ \mathcal{E}_n) \Rightarrow\!\!\!\!\Rightarrow_\beta \mathcal{S}_\iota$

7 Specialization with Mutable Store

If we set $\mathrm{T}\ \tau = (\tau \times \mathrm{Store} \to \mathrm{Code} \times \mathrm{Store}) \to \mathrm{Store} \to \mathrm{Code} \times \mathrm{Store}$ (a continuation-passing and store-passing monad) we obtain a specializer for a language with mutable store [17]. The expansion now reads as follows (using $(,)$ for pairing and π_i for projection).

$$\begin{array}{ll}
\mathcal{G}_s[\![x]\!] & \equiv x \\
\mathcal{G}_s[\![\lambda x.M]\!] & \equiv \lambda x.\mathcal{G}_s[\![M]\!] \\
\mathcal{G}_s[\![\mathrm{apply}\ M_1\ M_2]\!] & \equiv \lambda ks.\mathcal{G}_s[\![M_1]\!]@\mathcal{G}_s[\![M_2]\!]@k@s \\
\mathcal{G}_s[\![\hat{\lambda}(x, M)]\!] & \equiv \hat{\lambda}(x, \mathcal{G}_s[\![M]\!]) \\
\mathcal{G}_s[\![\hat{@}(M_1, M_2)]\!] & \equiv \hat{@}(\mathcal{G}_s[\![M_1]\!], \mathcal{G}_s[\![M_2]\!]) \\
\mathcal{G}_s[\![\mathrm{unit}(M)]\!] & \equiv \lambda ks.k@(\mathcal{G}_s[\![M]\!], s) \\
\mathcal{G}_s[\![\mathrm{let}\ x \Leftarrow M_1\ \mathrm{in}\ M_2]\!] & \equiv \lambda ks.\mathcal{G}_s[\![M_1]\!]@(\lambda(x, s').\mathcal{G}_s[\![M_2]\!]@k@s')@s \\
\mathcal{G}_s[\![\sharp_{\mathrm{Code}}(M)]\!] & \equiv \pi_1(\mathcal{G}_s[\![M]\!]@(\lambda(x, s).(x, s))@s_{\mathrm{empty}}) \\
\mathcal{G}_s[\![\mathrm{shift}_{\mathrm{Code}}\ x.M]\!] & \equiv \lambda ks.(\lambda x.\mathcal{G}_s[\![M]\!]@(\lambda(z, s').(z, s'))@s)@\lambda y.\lambda k's'.k'@(k@(y, s'))
\end{array}$$

The only problematic cases are $\sharp_{\mathrm{Code}}(M)$ and $\mathrm{shift}_{\mathrm{Code}}\ x.M$. The $\sharp_{\mathrm{Code}}(M)$ case shows that the standard reset operator is not sufficient here. The computation M is applied to the empty continuation $\lambda(x, s).(x, s)$ and the empty store s_{empty}. In the end the Store component is discarded and only the value returned. An implementation of reset would have to thread the store through the computation.

Since $\sharp_{\mathrm{Code}}(M)$ discards the static store, the binding-time analysis *must guarantee* that each reference, whose lifetime crosses the effect delimiter, is dynamic. We have developed such an analysis elsewhere [37].

To understand the implementation of $\mathrm{shift}_{\mathrm{Code}}\ x.M$ observe that the computation M is applied to the empty continuation and to the current store s. In the translated term, each occurrence of x stands for a function that accepts a value y, a continuation k', and a store s'. The function applies the captured continuation k to (y, s') to obtain the result of k paired with the resulting store. Both are passed to k' to produce the result of the computation.

Specialization with a store is not very interesting without any operations on it. We add the standard set of primitive operations, to allocate, read, and update references, to the source language.

$$\begin{array}{l}
E ::= \ldots \mid \mathrm{ref}\ E \mid !\ E \mid E := E \mid \underline{\mathrm{ref}}\ E \mid \underline{!}\ E \mid E \underline{:=} E \\
\tau\ ::= \ldots \mid \mathrm{ref}\ \tau \mid \underline{\mathrm{ref}}\ \tau
\end{array}$$

The typing rules and operational semantics are again standard (similar to ML). The metalanguage and the implementation language are extended by the same set of operators, but the result type of all these operations is a computation type, i.e., we consider Store an abstract datatype with operations mkref : $\mathcal{M}_v[\![\tau \to \mathrm{ref}\ \tau]\!]$, rdref : $\mathcal{M}_v[\![\mathrm{ref}\ \tau \to \tau]\!]$, and wrref : $\mathrm{ref}\ \tau \to \mathcal{M}_v[\![\tau \to \tau]\!]$. Extending the explication is standard:

$$\begin{array}{ll}
\mathcal{E}_v(\mathrm{ref}\ E) & \equiv \mathrm{let}\ x \Leftarrow \mathcal{E}_v(E)\ \mathrm{in}\ \mathrm{ref}\ x \\
\mathcal{E}_v(!\ E) & \equiv \mathrm{let}\ x \Leftarrow \mathcal{E}_v(E)\ \mathrm{in}\ !\ x \\
\mathcal{E}_v(E_1 := E_2) & \equiv \mathrm{let}\ x_1 \Leftarrow \mathcal{E}_v(E_1)\ \mathrm{in}\ \mathrm{let}\ x_2 \Leftarrow \mathcal{E}_v(E_2)\ \mathrm{in}\ x_1 := x_2
\end{array}$$

$\mathcal{G}$ requires some more care.

$$\begin{array}{lll}
\mathcal{G}\llbracket \text{ref } M \rrbracket \equiv \text{ref } \mathcal{G}\llbracket M \rrbracket & \mathcal{G}\llbracket \underline{\text{ref}}\ M \rrbracket & \equiv \text{unit}(\widehat{\text{ref}}(\sharp_{\text{Code}}(\mathcal{G}\llbracket M \rrbracket))) \text{ if } M : \underline{\text{T}}\ \tau \\
\mathcal{G}\llbracket !\ M \rrbracket \equiv !\ \mathcal{G}\llbracket M \rrbracket & & \equiv \text{unit}(\widehat{\text{ref}}(\mathcal{G}\llbracket M \rrbracket)) \text{ otherwise} \\
\mathcal{G}\llbracket \underline{!}\ M \rrbracket \equiv \text{unit}(\hat{!}(\mathcal{G}\llbracket M \rrbracket)) & \mathcal{G}\llbracket M_1 := M_2 \rrbracket & \equiv \mathcal{G}\llbracket M_1 \rrbracket := \mathcal{G}\llbracket M_2 \rrbracket \\
& \mathcal{G}\llbracket M_1 \underline{:=} M_2 \rrbracket & \equiv \text{unit}(\widehat{:=}(\mathcal{G}\llbracket M_1 \rrbracket, \sharp_{\text{Code}}(\mathcal{G}\llbracket M_2 \rrbracket))) \text{ if } M_2 : \underline{\text{T}}\ \tau \\
& & \equiv \text{unit}(\widehat{:=}(\mathcal{G}\llbracket M_1 \rrbracket, \mathcal{G}\llbracket M_2 \rrbracket)) \text{ otherwise}
\end{array}$$

The translation depends on the type of terms. In the image of the call-by-name translation, the argument of $\underline{\text{ref}}$ and the second argument of $\underline{:=}$ have computation type. The placement of the delimiter $\sharp_{\text{Code}}()$ is required to preserve type correctness. Since each effect delimiter causes less effects to be performed at specialization time, it is obvious that in a call-by-name language with effects less computations are performed in a known context. The monadic expansion to the implementation language is straightforward.

$$\begin{array}{llll}
\mathcal{G}_s\llbracket \text{ref } M \rrbracket & \equiv \lambda ks.\text{mkref } \mathcal{G}_s\llbracket M \rrbracket\ ks & \mathcal{G}_s\llbracket \widehat{\text{ref}}(M) \rrbracket & \equiv \widehat{\text{ref}}(\mathcal{G}_s\llbracket M \rrbracket) \\
\mathcal{G}_s\llbracket !\ M \rrbracket & \equiv \lambda ks.\text{rdref } \mathcal{G}_s\llbracket M \rrbracket\ ks & \mathcal{G}_s\llbracket \hat{!}(M) \rrbracket & \equiv \hat{!}(\mathcal{G}_s\llbracket M \rrbracket) \\
\mathcal{G}_s\llbracket M_1 := M_2 \rrbracket & \equiv \lambda ks.\text{wrref } \mathcal{G}_s\llbracket M_1 \rrbracket\ \mathcal{G}_s\llbracket M_2 \rrbracket\ ks & \mathcal{G}_s\llbracket \widehat{:=}(M_1, M_2) \rrbracket & \equiv \widehat{:=}(\mathcal{G}_s\llbracket M_1 \rrbracket, \mathcal{G}_s\llbracket M_2 \rrbracket)
\end{array}$$

8 Specialization with Exceptions

Finally, we embark on processing exceptions at specialization time. We use a model of exceptions with "raise E" and "E_1 handle E_2" constructs to raise and intercept exceptions and one fixed type "Exception" of exceptions. We assume that E_2 is a function that maps Exception to the same type as E_1.

$$E ::= \ldots \mid \text{raise } E \mid E \text{ handle } E \mid \underline{\text{raise}}\ E \mid E\ \underline{\text{handle}}\ E$$

The extension of the explication translation is standard:

$$\begin{array}{ll}
\mathcal{E}_v(\text{raise } E) & = \text{let } x \Leftarrow \mathcal{E}_v(E) \text{ in raise } x \\
\mathcal{E}_v(E_1 \text{ handle } E_2) & = \mathcal{E}_v(E) \text{ handle } \lambda x.\mathcal{E}_v(E_2 @ x)
\end{array}$$

The interesting part is the translation $\mathcal{G}$.

$$\begin{array}{ll}
\mathcal{G}\llbracket \text{raise } M \rrbracket & = \text{raise } \mathcal{G}\llbracket M \rrbracket \\
\mathcal{G}\llbracket M_1 \text{ handle } M_2 \rrbracket & = \mathcal{G}\llbracket M_1 \rrbracket \text{ handle } \mathcal{G}\llbracket M_2 \rrbracket \\
\mathcal{G}\llbracket \underline{\text{raise}}\ M \rrbracket & = \widehat{\text{raise}}(\mathcal{G}\llbracket M \rrbracket) \\
\mathcal{G}\llbracket M_1\ \underline{\text{handle}}\ M_2 \rrbracket & = \widehat{\text{handle}}(\sharp_{\text{Code}}(\mathcal{G}\llbracket M_1 \rrbracket), \mathcal{G}\llbracket M_2 \rrbracket)
\end{array}$$

In the definition of the monadic expansion $\mathcal{M}_e$ for the exception monad, we write $[f, g] : (A + B) \rightarrow C$ if $f : A \rightarrow C$ and $g : B \rightarrow C$. We use standard notation for the injection functions inl $: A \rightarrow A + B$ and inr $: B \rightarrow A + B$. We do not use the straightforward exception monad, but a composition with the continuation monad, which is required to define a sound call-by-value specializer with exceptions.

$$\text{T}\ \tau = ((\tau + \text{Exception}) \rightarrow (\text{Code} + \text{Exception})) \rightarrow (\text{Code} + \text{Exception})$$

$$\begin{array}{ll}
\mathcal{G}_e[\![x]\!] & \equiv x \\
\mathcal{G}_e[\![\lambda x.M]\!] & \equiv \lambda x.\mathcal{G}_e[\![M]\!] \\
\mathcal{G}_e[\![M_1 @ M_2]\!] & \equiv \lambda k.\mathcal{G}_e[\![M_1]\!] @ \mathcal{G}_e[\![M_2]\!] @ k \\
\mathcal{G}_e[\![\hat{\lambda}(x, M)]\!] & \equiv \hat{\lambda}(x, \mathcal{G}_e[\![M]\!]) \\
\mathcal{G}_e[\![\hat{@}(M_1, M_2)]\!] & \equiv \hat{@}(\mathcal{G}_e[\![M_1]\!], \mathcal{G}_e[\![M_2]\!]) \\
\mathcal{G}_e[\![\mathrm{unit}(M)]\!] & \equiv \lambda k.k @ (\mathrm{inl}\ \mathcal{G}_e[\![M]\!]) \\
\mathcal{G}_e[\![\mathrm{let}\ x \Leftarrow M_1\ \mathrm{in}\ M_2]\!] & \equiv \lambda k.\mathcal{G}_e[\![M_1]\!] @ [\lambda x.\mathcal{G}_e[\![M_2]\!] @ k, \lambda y.k @ \mathrm{inr}\ y] \\
\mathcal{G}_e[\![\sharp_{\mathrm{Code}}(M)]\!] & \equiv \mathcal{G}_e[\![M]\!] @ [\lambda x.x, \lambda y.\hat{\lambda}(z, z)] \\
\mathcal{G}_e[\![\mathrm{shift}_{\mathrm{Code}}\ x.M]\!] & \equiv \lambda k.(\lambda x.\mathcal{G}_e[\![M]\!] @ [\lambda z.\mathrm{inl}\ z, \lambda z.\mathrm{inr}\ z]) @ \lambda y.\lambda k'.k' @ (k @ (\mathrm{inl}\ y)) \\
\mathcal{G}_e[\![\mathrm{raise}\ M]\!] & \equiv \lambda k.k @ (\mathrm{inr}\ \mathcal{G}_e[\![M]\!]) \\
\mathcal{G}_e[\![\widehat{\mathrm{raise}}(M)]\!] & \equiv \widehat{\mathrm{raise}}(\mathcal{G}_e[\![M]\!]) \\
\mathcal{G}_e[\![M_1\ \mathrm{handle}\ M_2]\!] & \equiv \lambda k.\mathcal{G}_e[\![M_1]\!] @ [\lambda x.k @ (\mathrm{inl}\ x), \mathcal{G}_e[\![M_2]\!] @ k] \\
\mathcal{G}_e[\![M_1\ \mathrm{handle}\ M_2]\!] & \equiv \mathcal{G}_e[\![M_1]\!]\ \mathrm{handle}\ \mathcal{G}_e[\![M_2]\!]
\end{array}$$

Again, the placement of $\sharp_{\mathrm{Code}}(M)$ restricts the binding-time analysis. All exceptions that may cross the effect delimiter must be dynamic.

9 Related Work

There are two formalizations of partial evaluation using Moggi's work [30]. Hatcliff and Danvy [21] define a binding-time analysis and specialization for an annotated version of the monadic metalanguage. Their specializer exploits the monadic law 3 to "flatten" nested let expressions. To obtain an executable specification of the specializer they define a separate operational semantics (close to an abstract machine) that they prove equivalent to their first definition of specialization. Lawall and Thiemann [26] define an annotated version of Moggi's computational lambda calculus and show that it is implementable through a annotated CPS translation. This translation forms a reflection in a annotated lambda calculus of the annotated computational lambda calculus. Thus they show that a particular flavor of continuation-based partial evaluation is sound for all monadic models, thereby establishing firm ground for the development of specializers with computational effects expressed through monads.

In contrast to these works, the present work continues the work on monadic interpreters [18, 28, 40] in that it shows how to use monads to structure specializers in functional programming languages. Hence, the focus is on directly executable specifications. Our specifications implement the flattening transformation (the monad law 3) using special operations of the monad used to implement the specializer. Incidentally, these operations correspond to effect delimiters, control operators, and store operators [22]. We rely on the two above works [21, 26] for the soundness of this approach.

Effect delimiters have been considered by a number of researchers for varying purposes. Riecke and Viswanathan [35, 36] construct fully abstract denotational semantics for languages with monadic effects. Launchbury and Peyton Jones [24] define an effect delimiter for the state monad with a second order polymorphic type to encapsulate state-based computations. Similar operators have been used by Dussart et al [16] in order to get satisfactory results in a type specializer for the monadic metalanguage extended with mutable store (as in Sec.7).

There are offline partial evaluators for first-order imperative languages [1, 5, 7, 11, 12] and for higher-order languages [2, 3, 8, 29]. However, most partial evaluators for higher-order imperative languages [2, 3, 6] defer all computational effects to run time [5]. Realistic partial evaluators for higher-order languages with side effects must be able to perform side effects at specialization time. The only partial evaluator so far capable of this has been specified for a subset of Scheme by Thiemann and Dussart [17]. That work defines the specializer in extended continuation-passing store-passing style, it defines a binding-time analysis (which is proved correct elsewhere [37]), and considers pragmatic aspects such as efficient management of the store at specialization time and specialization of named program points. In contrast, the present work identifies a general scheme underlying the construction of specializers that address languages with computational effects. It does so in an evaluation-order independent framework and is built around monads in order to achieve maximum flexibility.

Birkedal and Welinder [2] developed an ad hoc scheme to deal with exceptions in their specializer for ML. It needs a separate correctness proof, because it is not based on the monadic metalanguage.

10 Conclusion

We have specified a generic framework for partial evaluation and demonstrated that it subsumes many existing algorithms for partial evaluation. The framework is correct for all monadic instances since it only performs reductions which are sound in the computational metalanguage.

In addition, we have investigated the construction of a generic binding-time analysis for languages with arbitrary effects, the construction of program generators instead of specializers, and the construction of specializers in direct style. These are reported in the full version of this paper [38] which also considers the call-by-name explication and some further language constructs, namely numbers and primitive operations, conditionals, and recursion.

References

1. Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).
2. Lars Birkedal and Morten Welinder. Partial evaluation of Standard ML. Rapport 93/22, DIKU, University of Copenhagen, 1993.
3. Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Programming,* 17:3–34, 1991.
4. Anders Bondorf. Improving binding times without explicit CPS-conversion. In *Proc. 1992 ACM Conference on Lisp and Functional Programming,* pages 1–10, San Francisco, California, USA, June 1992.

5. Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Programming*, 16(2):151–195, 1991.
6. Anders Bondorf and Jesper Jørgensen. Efficient analysis for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, July 1993.
7. Mikhail A. Bulyonkov and Dmitrij V. Kochetov. Practical aspects of specialization of Algol-like programs. In Danvy et al. [15], pages 17–32.
8. Charles Consel. Polyvariant binding-time analysis for applicative languages. In David Schmidt, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '93*, pages 66–77, Copenhagen, Denmark, June 1993. ACM Press.
9. Charles Consel and Olivier Danvy. For a better support of static data flow. In John Hughes, editor, *Proc. Functional Programming Languages and Computer Architecture 1991*, volume 523 of *Lecture Notes in Computer Science*, pages 496–519, Cambridge, MA, 1991. Springer-Verlag.
10. Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In POPL1993 [33], pages 493–501.
11. Charles Consel, Luke Hornof, François Noël, Jacques Noyé, and Nicolae Volanschi. A uniform approach for compile-time and run-time specialization. In Danvy et al. [15], pages 54–72.
12. Charles Consel and Francois Noël. A general approach for run-time specialization and its application to C. In *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg, Fla., January 1996. ACM Press.
13. Olivier Danvy. Back to direct style. *Science of Programming*, 22:183–195, 1994.
14. Olivier Danvy and Andrzej Filinski. Abstracting control. In LFP 1990 [27], pages 151–160.
15. Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Dagstuhl Seminar on Partial Evaluation 1996*, volume 1110 of *Lecture Notes in Computer Science*, Schloß Dagstuhl, Germany, February 1996. Springer-Verlag.
16. Dirk Dussart, John Hughes, and Peter Thiemann. Type specialisation for imperative languages. In Mads Tofte, editor, *Proc. International Conference on Functional Programming 1997*, pages 204–216, Amsterdam, The Netherlands, June 1997. ACM Press, New York.
17. Dirk Dussart and Peter Thiemann. Partial evaluation for higher-order languages with state. Berichte des Wilhelm-Schickard-Instituts WSI-97-XX, Universität Tübingen, April 1997.
18. David Espinosa. Building interpreters by transforming stratified monads. ftp://altdorf.ai.mit.edu/pub/dae, June 1994.
19. Carsten K. Gomard and Neil D. Jones. Compiler generation by partial evaluation: A case study. *Structured Programming*, 12:123–144, 1991.
20. John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In *Proc. 21st Annual ACM Symposium on Principles of Programming Languages*, pages 458–471, Portland, OG, January 1994. ACM Press.
21. John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7(5):507–542, 1997.
22. G. F. Johnson and Dominic Duggan. Stores and partial continuations as first-class objects in a language and its environment. In *Proc. 15th Annual ACM Symposium on Principles of Programming Languages*, pages 158–168, San Diego, California, January 1988. ACM Press.

23. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.
24. John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *Proc. of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 24–35, Orlando, Fla, USA, June 1994. ACM Press.
25. Julia Lawall and Olivier Danvy. Continuation-based partial evaluation. In *Proc. 1994 ACM Conference on Lisp and Functional Programming*, pages 227–238, Orlando, Florida, USA, June 1994. ACM Press.
26. Julia Lawall and Peter Thiemann. Sound specialization in the presence of computational effects. In *Proc. Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*, pages 165–190, Sendai, Japan, September 1997. Springer-Verlag.
27. *Proc. 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, 1990. ACM Press.
28. Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In POPL1995 [34], pages 333–343.
29. Karoline Malmkjær, Nevin Heintze, and Olivier Danvy. ML partial evaluation using set-based analysis. In *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications*, number 2265 in INRIA Research Report, pages 112–119, Orlando, Florida, June 1994.
30. Eugenio Moggi. Notions of computations and monads. *Information and Computation*, 93:55–92, 1991.
31. Bàrbara Moura, Charles Consel, and Julia Lawall. Bridging the gap between functional and imperative languages. Publication interne 1027, Irisa, Rennes, France, July 1996.
32. Simon L. Peyton Jones and Philip L. Wadler. Imperative functional programming. In POPL1993 [33], pages 71–84.
33. *Proc. 20th Annual ACM Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 1993. ACM Press.
34. *Proc. 22nd Annual ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 1995. ACM Press.
35. John Riecke. Delimiting the scope of effects. In Arvind, editor, *Proc. Functional Programming Languages and Computer Architecture 1993*, pages 146–155, Copenhagen, Denmark, June 1993. ACM Press, New York.
36. John Riecke and Ramesh Viswanathan. Isolating side effects in sequential languages. In POPL1995 [34], pages 1–12.
37. Peter Thiemann. Correctness of a region-based binding-time analysis. In *Proc. Mathematical Foundations of Programming Semantics, Thirteenth Annual Conference*, volume 6 of *Electronic Notes in Theoretical Computer Science*, page 26, Pittsburgh, PA, March 1997. Carnegie Mellon University, Elsevier Science BV. URL: http://www.elsevier.nl/locate/entcs/volume6.html.
38. Peter Thiemann. A generic framework for specialization. Berichte des Wilhelm-Schickard-Instituts WSI-97-XXX, Universität Tübingen, October 1997.
39. Philip L. Wadler. Comprehending monads. In LFP 1990 [27], pages 61–78.
40. Philip L. Wadler. The essence of functional programming. In *Proc. 19th Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 1992. ACM Press.