Imperative Functional Specialization

Dirk Dussart, Peter Thiemann

WSI-96-28

Wilhelm-Schickard-Institut für Informatik
Universität Tübingen
Sand 13
D-72076 Tübingen
Germany

Tel.: x49 7071 29-75467
Fax: x49 7071 29-5958
Email: thiemann@informatik.uni-tuebingen.de

©Wilhelm-Schickard-Institut 1996
ISSN 0946-3852

Imperative Functional Specialization

Dirk Dussart
Department of Computer Science
K.U.Leuven*
Dirk.Dussart@cs.kuleuven.ac.be

Peter Thiemann
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen†
thiemann@informatik.uni-tuebingen.de

Abstract

We present an offline partial evaluator for a higher-order subset of ML with reference types. Its novel feature is the ability to perform assignments to local and global variables at specialization time. The presented technology applies directly to realistic programming languages, such as Scheme and Standard ML. It is an instance of a novel monad-based partial-evaluation framework, which generalizes all known approaches to partial evaluation for functional languages to date. The particular instance here is a continuation/store monad which provides the necessary control and store operators. Store operators are implemented efficiently by means of first-class stores.

To render the partial evaluator completely automatic we use program analyses to a great extent. Specifically, the traditional binding-time analysis is supported by an effect analysis and a reaching-definitions analysis. All analyses are cast in the framework of type-based and effect-based program analysis.

The correctness of the binding-time analysis is established relative to the correctness of a binding-time analysis for the lambda calculus using monadic translations.

Keywords: higher-order programming, program transformation, partial evaluation, state

1 Introduction

Partial evaluation [46, 16] is an automatic program-specialization technique. It trades generality for efficiency by constructing a specialized program from a source program and its known (*static*) input. The specialized program takes the remaining (*dynamic*) input and delivers the same result as the source program applied to the whole input, but in a more efficient way. We consider the *offline* variant of partial evaluation which separates the transformation in two stages: *binding-time analysis* and specialization proper. The binding-time analysis (BTA) processes the source program and the binding times (static/dynamic) of its arguments. It produces an annotated program with all expressions that solely depend on the static input marked static and all the remaining expressions marked dynamic. This simplifies the specializer to an interpreter for annotated programs, which reduces static expressions and generates code for dynamic expressions.

Offline partial evaluators exist for first-order imperative languages [10, 3, 13, 18, 17] as well as for higher-order languages [8, 14, 7, 54]. Current partial evaluators for higher-order imperative languages [8, 11, 7] defer all assignments and other side-effecting operations (such as I/O) to run time [10]. Realistic partial evaluators for higher-order languages with side effects must be able to perform side effects at specialization time.

As an example, consider the code in Fig. 1 defining a class of counter objects using a procedural encoding. Partial evaluators such as Similix [8] or SML-Mix [7] defer all computations involving a

*Celestijnenlaan 200A, B-3001 Leuven, Belgium. Supported by the National Fund for Scientific Research Belgium (N.F.W.O.). This work was initiated during a visit at Tübingen University funded by a grant of the Ministerium für Wissenschaft und Forschung.

†Sand 13, D-72076 Tübingen, Germany.

```
let val counter_class =
    (fn () => let val slot = ref 0
                  val mset = (fn x => slot := x)
                  val mget = (fn () => !slot )
                  val madd = (fn x => slot := !slot + x)
              in {get = mget, set = mset, add = madd}
              end)
in let val cnt = counter_class ()
   in ((#set cnt) 21;
       (#add cnt) ((#get cnt) ());
       (#get cnt) ())
   end
end
```

Figure 1: Counter Program

```
let val slot = ref 0
in  slot := 21;
    let val v = !slot
    in  slot := !slot+v
    end;
    !slot
end
```

Figure 2: Residual Counter Program

counter object to run time. Fig. 2 shows the resulting residual program. In contrast, our partial evaluator reduces the source expression to 42. This message-passing style of programming comes close to object-oriented programming because it enforces data abstraction and representation independence [1, 34]. To the best of our knowledge, our specializer is the first to achieve satisfactory specialization of programs with higher-order functions and local state. Hence, it is the first offline specializer which is applicable to object-oriented programming techniques.

Specialization of a higher-order language with assignments raises interesting semantic issues. As the specializer performs some assignments statically, it reorders the assignments with respect to standard evaluation. The specializer may even perform assignments that never take place during standard evaluation. A large part of this work is concerned with the discovery of such independence properties. Despite these problems we are able to define a partial evaluator which achieves satisfactory specialization.

1.1 Specialization

Denotational semantics [73] suggests one way to perform offline partial evaluation for a higher-order language with side effects: Convert the source programs to purely functional programs in which the store is made explicit (for example, using a transformation to continuation-passing store-passing style, CPSPS) and apply a standard partial evaluator [61]. This approach is related to the idea of transforming programs to continuation-passing style (CPS) to improve their static data flow [15]. Both transformations make the monad of program execution [58, 80, 81] explicit. We call this approach "monadic expansion" because it amounts to rewriting the program in monadic style and expanding the definitions of the operations of the monad. There are some problems with this approach:

1. Monadic expansion usually introduces many extra higher-order functions. Specialising programs in explicit monadic style results in specialized programs written in the same style.

As most language implementations and specializers are geared towards efficient execution of "typical programs" (usually synonymous with direct-style programs that do not use higher-order functions extensively), we expect some slowdown in the specialization itself and in the specialized programs (in explicit monadic style), too [49].

2. Offline partial evaluators subject their input to a BTA. The results of the analysis provide useful feedback on the degree of specialization to the user. It is hard to convey the results of such an analysis on a program in explicit monadic style.

3. Using the straightforward monadic expansion does not immediately lead to satisfactory binding times.

Therefore, we pursue another way to attack the problem. As specialization involves interpretation, it appears natural not to perform monadic expansion on source programs, but instead to write the specializer in monadic style. This has been done for the continuation monad [9]. The resulting improvement of the static data flow is equivalent to that obtained by converting source programs to CPS. Moreover, the specialized program is still in direct style.

Analogously, we avoid translating source programs to explicit monadic style by moving the monad into the specializer, as has been demonstrated for monadic interpreters [81, 32, 52]. In our specific instance, we avoid translating source programs to CPSPS by writing the specializer using a continuation and store monad. We call the resulting specialization technology *continuation/store-based partial evaluation.* We take the analogy one step further: Lawall and Danvy [49] show how to write a continuation-based specializer in direct style with control operators [22, 23]. These control operators are special operations of the continuation monad. In our setting, we have also written the specializer in direct style using the same control operators and some non-standard store operators [45].

1.2 Binding-Time Analysis

Non-standard and annotated type systems can serve as specifications for BTAs [37, 40, 27, 36]. Starting from a standard type derivation, an annotated type system adds (binding time) annotations to the type language and defines well-formedness criteria for (binding time) annotated types and binding-time type-derivations. Interpreting the typing rules as constraints on the annotations leads directly to a BTA algorithm.

This work extends previous work by taking an effect system as the basis for a BTA specification. The effect system determines for each abstraction a set of references that may be read or written during all possible applications of the abstraction. The BTA enforces—in addition to the usual constraints [27, 36]—that all global references accessed (read or written) by a dynamic abstraction are *dynamic.*

The binding-time annotations induce splitting the store into a dynamic part and many static fragments—one for each dynamic abstraction. This leads naturally to a reinterpretation of the specializer over a variant of the continuation/store monad in which the store splits up in two stores. The first step in the correctness proof of the BTA consists of showing the equivalence between the semantics over this continuation/double-store monad and the semantics over the continuation/store monad. In the second step we show that a binding-time directed monadic expansion of annotated programs yields a well-annotated purely functional program. The correctness of the BTA then follows by appeal to the correctness of BTAs for the lambda calculus [63, 82, 38].

1.3 Contributions

We have defined a novel *monad-based partial evaluation* framework that generalizes all existing partial evaluators for functional programming languages. The framework gives generic recipes for specializing programs over a monad:

- perform monadic expansion of the source program and apply a standard specializer,

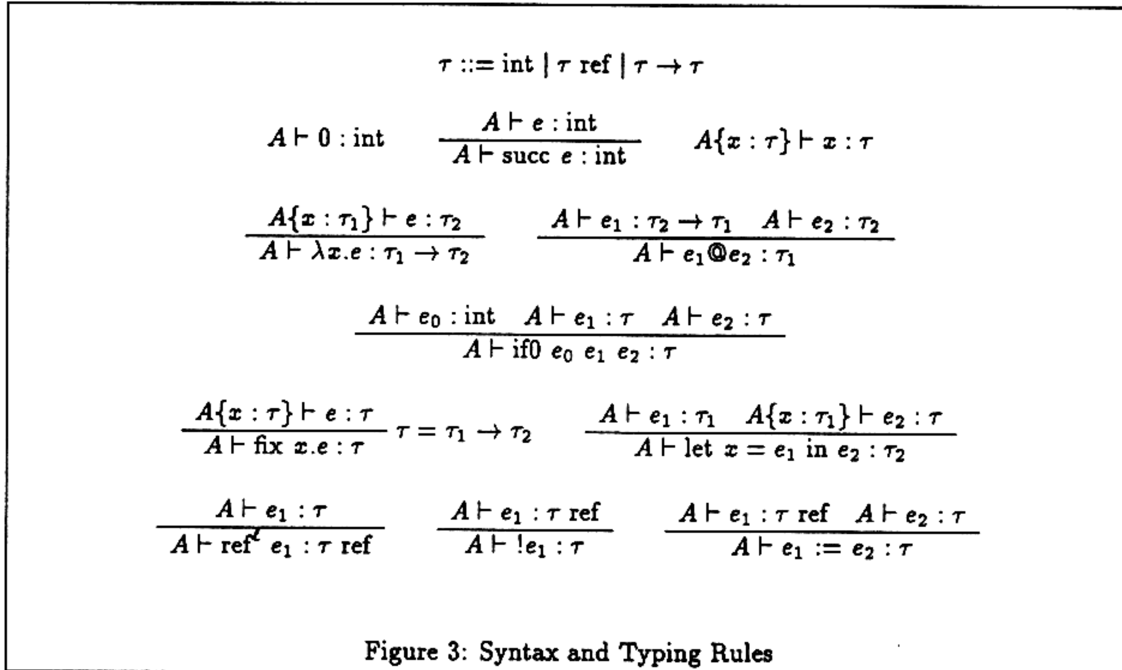

Figure 3: Syntax and Typing Rules

- write the specializer in monadic style, or
- write the specializer in direct style with monadic operations.

We also give a general recipe to construct monadic PGGs (program generator generators or cogens).

For the continuation/store monad we obtain an offline partial evaluator for a higher-order imperative language that performs assignments at specialization time. It is written in direct style with control and store operators. This settles a longstanding open problem in the field.

A type-based BTA renders the partial evaluator completely automatic. The novel aspect of our BTA is the inclusion of effect information.

We have proved the correctness of our partial evaluator by means of an annotation-preserving transformation to monadic style.

1.4 Overview

In Sec. 2 we define the language λ^{ML} and its monadic semantics. The monadic semantics forms the foundation of the specializer in Sec. 3. Sec. 4 outlines the BTA and Sec. 5 deals with its correctness proof. Sec. 6 introduces the idea of reaching-definitions analysis. Sec. 7 discusses some further issues, i.e., dynamic side effects, partially static data, direct style implementation of the continuation and store monad, and designing a PGG using combinators. Finally, we discuss related work (Sec. 8), conclude, and sketch possible further work.

2 Language and Semantics

We consider λ^{ML}—a simply typed call-by-value lambda calculus which includes references. Fig. 3 defines syntax and standard typing judgements $A \vdash e : \tau$ (read: Under the set of type assumptions A expression e has type τ) of λ^{ML}. In addition to the standard constructions there is the constant 0 and the primitive operation succ e (successor). Furthermore we can create references ref e, dereference references $!e$, and assign to references $e_1 := e_2$. Each (sub-) expression carries a unique label $l \in \text{Lab}$ which we indicate with a superscript when necessary.

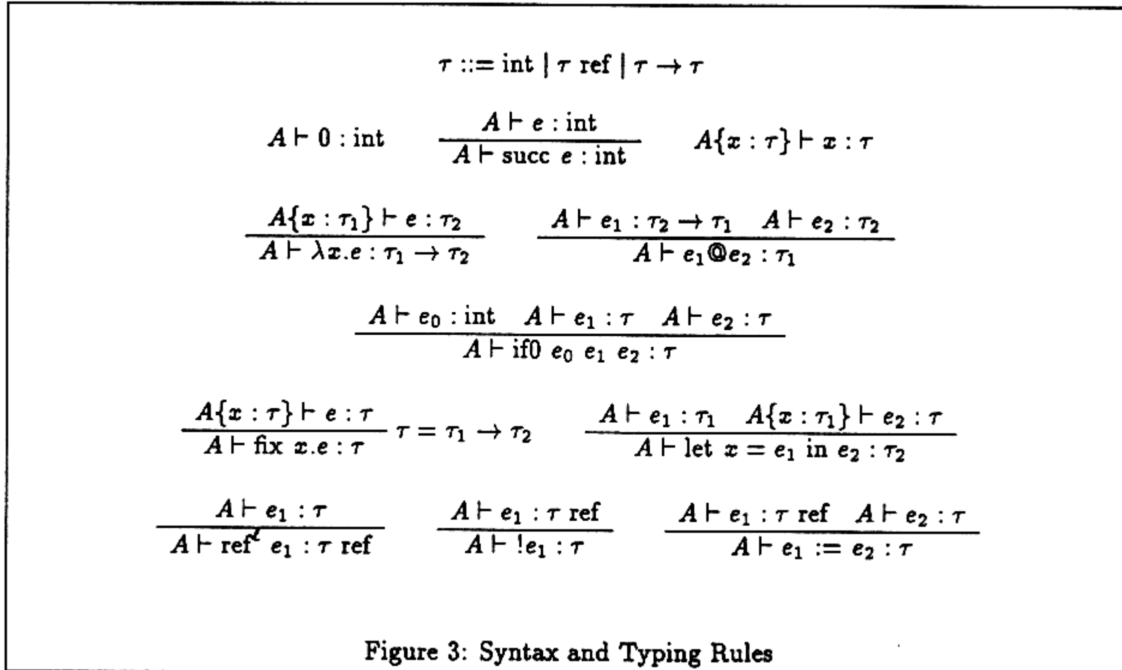

Figure 3: Syntax and Typing Rules

- write the specializer in monadic style, or
- write the specializer in direct style with monadic operations.

We also give a general recipe to construct monadic PGGs (program generator generators or cogens).

For the continuation/store monad we obtain an offline partial evaluator for a higher-order imperative language that performs assignments at specialization time. It is written in direct style with control and store operators. This settles a longstanding open problem in the field.

A type-based BTA renders the partial evaluator completely automatic. The novel aspect of our BTA is the inclusion of effect information.

We have proved the correctness of our partial evaluator by means of an annotation-preserving transformation to monadic style.

1.4 Overview

In Sec. 2 we define the language λ^{ML} and its monadic semantics. The monadic semantics forms the foundation of the specializer in Sec. 3. Sec. 4 outlines the BTA and Sec. 5 deals with its correctness proof. Sec. 6 introduces the idea of reaching-definitions analysis. Sec. 7 discusses some further issues, i.e., dynamic side effects, partially static data, direct style implementation of the continuation and store monad, and designing a PGG using combinators. Finally, we discuss related work (Sec. 8), conclude, and sketch possible further work.

2 Language and Semantics

We consider λ^{ML}—a simply typed call-by-value lambda calculus which includes references. Fig. 3 defines syntax and standard typing judgements $A \vdash e : \tau$ (read: Under the set of type assumptions A expression e has type τ) of λ^{ML}. In addition to the standard constructions there is the constant 0 and the primitive operation succ e (successor). Furthermore we can create references ref e, dereference references $!e$, and assign to references $e_1 := e_2$. Each (sub-) expression carries a unique label $l \in \mathrm{Lab}$ which we indicate with a superscript when necessary.

$$
\begin{array}{lcl}
\mathcal{E} : \text{Expr} \to \text{Env} \to M \ \text{Val} & & \\
\rho \in \text{Env} = \text{Var} \to \text{Val} & & \\
\mathcal{E}[\![0]\!]\rho & = & \eta(0) \\
\mathcal{E}[\![\text{succ } e_1]\!]\rho & = & \mathcal{E}[\![e_1]\!]\rho \Rrightarrow (\lambda y.\eta(y+1)) \\
\mathcal{E}[\![x]\!]\rho & = & \eta(\rho(x)) \\
\mathcal{E}[\![\lambda x.e]\!]\rho & = & \eta(\lambda y.\mathcal{E}[\![e]\!]\rho[y/x]) \\
\mathcal{E}[\![e_1 @ e_2]\!]\rho & = & \mathcal{E}[\![e_1]\!]\rho \Rrightarrow (\lambda f.\mathcal{E}[\![e_2]\!]\rho \Rrightarrow \lambda y.fy) \\
\mathcal{E}[\![\text{fix } x.e]\!]\rho & = & \text{fixM } \lambda y.\mathcal{E}[\![e]\!]\rho[y/x] \\
\mathcal{E}[\![\text{let } x = e_1 \text{ in } e_2]\!]\rho & = & \mathcal{E}[\![e_1]\!]\rho \Rrightarrow (\lambda y.\mathcal{E}[\![e_2]\!]\rho[y/x]) \\
\mathcal{E}[\![\text{if0 } e_0 \ e_1 \ e_2]\!]\rho & = & \mathcal{E}[\![e_0]\!]\rho \Rrightarrow (\lambda x.\text{if } (x=0)(\mathcal{E}[\![e_1]\!]\rho)(\mathcal{E}[\![e_2]\!]\rho)) \\
\mathcal{E}[\![\text{ref}^{\ell} \ e]\!]\rho & = & \mathcal{E}[\![e]\!]\rho \Rrightarrow (\lambda y.\text{alloc } \ell \ y) \\
\mathcal{E}[\![!e]\!]\rho & = & \mathcal{E}[\![e]\!]\rho \Rrightarrow (\lambda p.\text{fetch } p) \\
\mathcal{E}[\![e_1 := e_2]\!]\rho & = & \mathcal{E}[\![e_1]\!]\rho \Rrightarrow (\lambda p.\mathcal{E}[\![e_2]\!]\rho \Rrightarrow (\lambda v.\text{assign } p \ v))
\end{array}
$$

Figure 4: Monadic Semantics

The denotational semantics is given in monadic style. It defines a call-by-value lambda-calculus where evaluation proceeds from left to right. This semantics forms the basis of our monadic specializer.

2.1 Monads

Monads help structure denotational semantics [58], build modular interpreters and compilers [32, 52, 51], structure functional programs [80, 81], and incorporate I/O and assignments into pure functional programming languages [64, 47, 48]. In Wadler's presentation [81], a monad is a parameterized datatype $M\ \alpha$ denoting a computation that delivers values of type α. A monad has two polymorphic operations

$$
\begin{array}{lcl}
\eta(_) & : & \alpha \to M\ \alpha \\
_ \Rrightarrow _ & : & M\ \alpha \to (\alpha \to M\ \beta) \to M\ \beta
\end{array}
$$

that satisfy the monad laws: $\Rrightarrow$ is associative with left and right unit $\eta()$. Intuitively, $\eta(x)$ is a trivial computation that returns the value x. The operation $m \Rrightarrow f$ constructs a computation that first performs the computation $m : M\ \alpha$, retrieves the result $x : \alpha$, and performs the computation $f\ x : M\ \beta$. In the identity monad, which corresponds to programs in direct style, we have $M\ \alpha = \alpha$, $\eta(x) = x$ the identity function, and $m \Rrightarrow f = fm$—function application.

2.2 Monadic Semantics

Figure 4 defines a monadic semantics for the language. The semantic function $\mathcal{E}$ maps expressions e and environments ρ to computations in monad M that deliver values $c \in M(\text{Val})$. Environments ρ map variables x to values $v \in \text{Val}$. As customary, we omit the injections and sum decompositions. We assume some operators from the underlying monad:

$$
\begin{array}{lcl}
\text{alloc} & : & \text{Lab} \to \text{Val} \to M\ \text{Loc} \\
\text{fetch} & : & \text{Loc} \to M\ \text{Val} \\
\text{assign} & : & \text{Loc} \to \text{Val} \to M\ \text{Val} \\
\text{fixM} & : & (\text{Val} \to M\ \text{Val}) \to M\ \text{Val}
\end{array}
$$

The operation "alloc ℓ v" allocates a new cell in the store, initializes it with v, and returns its location. The operation "fetch p" dereferences the store location p and "assign p v" overwrites location p by v and returns v.

$$
\begin{array}{rrlcl}
 & & \text{Val} & = & \text{N} + \text{Loc} + (\text{Val} \to M\ \text{Val}) \\
 & & M\ \alpha & = & ((\alpha \times \text{Store})_\bot \to \text{Answer}) \to \text{Store} \to \text{Answer} \\
 & & \text{Answer} & = & (\text{Val} \times \text{Store})_\bot \\
\sigma & \in & \text{Store} & = & \text{Loc} \to (\text{Val} + \{\text{unused}\}) \\
\alpha & \in & \text{Loc} & = & \text{Addr} \times \text{Lab} \\
\ell & \in & \text{Lab} & & \text{unspecified} \\
 & & \text{Addr} & & \text{unspecified}
\end{array}
$$

Figure 5: Semantic Domains

Figure 5 defines the semantic domains. The domain equations have a solution in the category of (unpointed) dcpos. N denotes the set of non-negative integers, $+$, $\times$, and $\to$ denote categorical sum, product, and exponentiation, $A_\bot$ denotes lifting with injection up $\in A \to A_\bot$, strict denotes the projection from standard function space into strict function space, cf. [60].

Technically, $M\ \alpha$ is the strict monad obtained by lifting the continuation monad over the store monad. The monadic operations are defined by

$$
\begin{array}{lcl}
\eta(x) & = & \lambda k\sigma.k(\text{up}(x,\sigma)) \\
m \Rrightarrow f & = & \lambda k\sigma.m(\text{strict}\lambda(y,\sigma).fyk\sigma)\sigma \\
 & & \\
\text{alloc}\ \ell\ y & = & \lambda k\sigma.k(\text{up}(\alpha, \sigma[\alpha \mapsto y]))\ \text{where}\ \sigma\alpha = \text{unused} \land \alpha = (m, \ell) \\
\text{assign}\ \alpha\ y & = & \lambda k\sigma.k(\text{up}(y, \sigma[\alpha \mapsto y])) \\
\text{fetch}\ \alpha & = & \lambda k\sigma.k(\text{up}(\sigma\alpha, \sigma)) \\
\text{fixM}\ f & = & \text{fix}\ \lambda h.f(\lambda z.h \Rrightarrow \lambda g.gz)
\end{array}
$$

3 The Specializer

The monadic semantics serves also as an interpreter for our language. Likewise, a semantics for an annotated language serves as a specializer. In the annotated version of λ^{ML}, each syntactic construct has a dynamic counterpart that generates code. Syntactically, we mark the dynamic constructs by a superscript ^{D}.

The domains change slightly to include expressions.

$$
\begin{array}{rrlcl}
 & & \text{PEVal} & = & \text{N} + \text{Loc} + (\text{PEVal} \to M\ \text{PEVal}) + \text{Expr} \\
\sigma & \in & \text{Store} & = & \text{Loc} \to (\text{PEVal} + \{\text{unused}\})
\end{array}
$$

Figure 6 shows the definition of the monadic specializer. We omit the static part which is identical to the monadic semantics (the interpreter) in Fig. 4. In the meta-language, we use underlining for code generation [62]. For example, $\underline{\lambda}(_).(_)$ accepts a variable name and code and constructs a lambda abstraction. The marker $^\diamond$ denotes a freshly generated variable. The operation *quote*(_) converts first-order values to their syntactic representation.

The specification uses some new monadic operations. Two of them, shift and reset, belong to the continuation monad [22, 23]. Shift abstracts the current context up to the next lexically enclosing reset in a function and discards it. We need the operators in a slightly modified form suitable for the continuation/store monad:

$$
\begin{array}{lcl}
\text{reset}\ (_) & : & M\ \text{Val} \to M\ \text{Val} \\
\text{shift} & : & ((\text{Val} \to M\ \text{Val}) \to M\ \text{Val}) \to M\ \text{Val} \\
\text{reset}\ (m) & = & \lambda k\sigma.k(m(\lambda z.z)\sigma) \\
\text{shift}\ f & = & \lambda k\sigma.f(\lambda v k'\sigma.k'(k(\text{up}(v,\sigma))))(\lambda z.z)\sigma
\end{array}
$$

The other pair of operators originates from the store monad [45]. curstore : M Store, reifies the current store and returns it as the result of the computation, while withstore $\sigma\ m : M\ \alpha$,

$$
\begin{array}{lcl}
\mathcal{S} : \text{Expr} \to \text{Env} \to M \text{ PEVal} & & \\
\rho \in \text{Env} = \text{Var} \to \text{PEVal} & & \\
\mathcal{S}[\![\text{lift } e]\!]\rho & = & \mathcal{S}[\![e]\!]\rho \Rightarrow \lambda y.\eta(\mathit{quote}(y)) \\
\mathcal{S}[\![\text{succ}^D\ e]\!]\rho & = & \mathcal{S}[\![e]\!]\rho \Rightarrow \lambda y.\eta(\underline{\text{succ}}\ y) \\
\mathcal{S}[\![\lambda^D x.e]\!]\rho & = & \text{reset } (\mathcal{S}[\![e]\!]\rho[x^\diamond/x]) \Rightarrow \lambda y.\eta(\underline{\lambda} x^\diamond.y) \\
\mathcal{S}[\![e_1 @^D e_2]\!]\rho & = & \mathcal{S}[\![e_1]\!]\rho \Rightarrow \lambda y_1.\mathcal{S}[\![e_2]\!]\rho \Rightarrow \lambda y_2.\eta(y_1 \underline{@} y_2) \\
\mathcal{S}[\![\text{fix}^D\ x.e]\!]\rho & = & \text{reset } (\mathcal{S}[\![e]\!]\rho[x^\diamond/x]) \Rightarrow \lambda y.\eta(\underline{\text{fix}}\ x^\diamond.y) \\
\mathcal{S}[\![\text{let}^D\ x = e_1 \text{ in } e_2]\!]\rho & = & \text{shift } \lambda j.\mathcal{S}[\![e_1]\!]\rho \Rightarrow \lambda y_1. \\
 & & \quad \text{reset } (\mathcal{S}[\![e_2]\!]\rho[x^\diamond/x] \Rightarrow j) \Rightarrow \lambda y_2. \\
 & & \quad \eta(\underline{\text{let}}\ x^\diamond = y_1\ \underline{\text{in}}\ y_2) \\
\mathcal{S}[\![\text{if0}^D\ e_1\ e_2\ e_3]\!]\rho & = & \text{shift } \lambda j.\mathcal{S}[\![e_1]\!]\rho \Rightarrow \lambda y_1. \\
 & & \quad \text{curstore} \Rightarrow \lambda s.\text{reset } (\mathcal{S}[\![e_2]\!]\rho \Rightarrow j) \Rightarrow \lambda y_2. \\
 & & \quad \text{withstore } s \text{ (reset } (\mathcal{S}[\![e_3]\!]\rho \Rightarrow j)) \Rightarrow \lambda y_3. \\
 & & \quad \eta(\underline{\text{if0}}\ y_1\ y_2\ y_3) \\
\mathcal{S}[\![\text{ref}^D\ e]\!]\rho & = & \mathcal{S}[\![e]\!]\rho \Rightarrow \lambda y.\eta(\underline{\text{ref}}\ y) \\
\mathcal{S}[\![!^D e]\!]\rho & = & \mathcal{S}[\![e]\!]\rho \Rightarrow \lambda y.\eta(\underline{!} y) \\
\mathcal{S}[\![e_1 :=^D e_2]\!]\rho & = & \mathcal{S}[\![e_1]\!]\rho \Rightarrow \lambda y_1.\mathcal{S}[\![e_2]\!]\rho \Rightarrow \lambda y_2.\eta(y_1 \underline{:=} y_2)
\end{array}
$$

Figure 6: Continuation/Store-Based Specializer

installs the store σ : Store during the computation $m : M\ \alpha$.

$$
\begin{array}{lcl}
\text{curstore} & = & \lambda k\sigma.k(\text{up}(\sigma, \sigma)) \\
\text{withstore } \sigma'\ m & = & \lambda k\sigma.m\ (\text{strict}\lambda(v, \sigma').k(\text{up}(v, \sigma)))\ \sigma'
\end{array}
$$

This pair of operations enables the specializer to perform static assignments in both branches of a dynamic conditional. After finishing the specialization of one branch and before entering the second branch, the specializer must reset the store to its state before entering the first branch. Also, two incomparable static stores may result from specializing the branches. Thus, the question is, in which store do we specialize the context. In our setting, continuation/store-based specialization comes to our rescue: It propagates the appropriate duplicated static context into the branches of the conditional such that we only need to specialize an empty context with respect to the merged store. Therefore, we do not need to worry about merging the stores.

4 Binding-Time Analysis

Specialization is more eager than standard evaluation. The specializer processes the body of a dynamic abstraction without knowing whether the abstraction is ever applied, where that may be, and how often that may happen. If the specializer statically accesses any global variable while processing the body of a dynamic abstraction this access will be out of the intended order. Hence, in addition to the usual consistency constraints [37, 27, 36] the BTA has to classify dynamic all global reference that may be accessed while processing the body of a dynamic abstraction. This ensures the right sequencing of all operations.

The BTA builds on an effect analysis which computes for each abstraction a set of references which may be read or written for all possible evaluations of the body of the abstraction. The effect analysis has two parts. The first part—*creation-point analysis*—connects values of reference type to their lexical creation points by decorating a standard type derivation. The effect derivation is then a decoration of creation-point derivations. Both the creation-point analysis and the effect analysis use subtyping and are proven sound with respect to an instrumented version of the denotational semantics. See our technical report [28] for full details.

The actual BTA starts by decorating effect derivations with binding-time annotations. Binding-time types are, therefore, annotated effect types. The rules are similar to those found in earlier work of the first author (cf. [27]), restricted to the subsystem without binding-time polymorphism. The key distinguishing feature is the rule for abstractions, which adds constraints on the annotations of global reference types in the type assumption. If the abstraction needs to be dynamic for some reason then these constraints ensure that all the global reference types that are possibly read or written according to the effect analysis are dynamic, too. Again, for lack of space we have to refer to the technical report [28] for full details.

Of course, the additional constraints can affect the binding times of the rest of the program and lead to overly conservative annotations. Fortunately, we can make up for that with a prepass that performs reference splitting based on a reaching-definitions analysis (see Sec. 6).

5 Correctness

In the correctness proof of the specializer we demonstrate that a well-annotated expression—according to the binding-time annotated effect system (EBTA)—does not cause the specializer to go wrong. The proof builds on the correctness of an existing specializer and BTA for the lambda calculus. We transform an EBTA well-annotated expression into a Lambdamix well-annotated expressions [38]. The transformation is monadic expansion for a suitable continuation-passing double-store-passing (CP2SPS) monad. This monad propagates two stores, one static store and one dynamic store. The monadic operators handle the static store differently than the dynamic store. Whereas the dynamic store is threaded through the whole computation, the life time of each static store is confined to a single dynamic abstraction.

The proof has two parts:

1. equivalence of the CP2SPS semantics and the CPSPS semantics and
2. the transformation to CP2SPS preserves well-annotatedness.

The next section provides an outline of the results, a companion technical report contains full details and proofs [28].

In the general case we need a separate store for values of each different binding-time annotated type. To make the proof feasible we assume that there are only references to integers, without losing generality.

5.1 Continuation-Passing Double-Store-Passing

The continuation-passing double-store-passing monad splits the store in two separate stores. Hence, the definition of the type constructor and the monadic operators:

$$
\begin{array}{lcl}
\mathcal{C}_{2s}(\text{PEVal}) & = & ((\text{PEVal} \times \text{DStore} \times \text{Store})_\bot \to \text{Answer}) \to \text{DStore} \to \text{Store} \to \text{Answer} \\
\text{Answer} & = & (\text{PEVal} \times \text{DStore} \times \text{Store})_\bot \\
\eta(x) & = & \lambda k\delta\sigma.k(\text{up}(x,\delta,\sigma)) \\
m \Rightarrow f & = & \lambda k\delta\sigma.m(\text{strict}\lambda(y,\delta,\sigma).fyk\delta\sigma)\delta\sigma \\
\text{withstore } s\ m & = & \lambda k\delta\sigma.m(\text{strict}\lambda(y,\delta,s').k(\text{up}(y,\delta,\sigma)))\delta s \\
\text{DStore} & = & \text{Store}
\end{array}
$$

"withstore s m" installs a new static store s for the duration of m and restores the previous static store afterwards. The dynamic store remains untouched. Using this monad, we construct an oblivious monadic interpreter for annotated programs that executes the annotated parts, too. However, it does not completely ignore the annotation, but uses it to process the store differently than the standard interpreter. The part for the static constructs remains the same as in Fig. 3. Fig. 7 shows the oblivious interpretation of the dynamic constructs. In the definition, σ_0 and $\sigma_\bot$ are both (different copies of) the empty store, i.e., $\lambda z.\text{unused}$. Whereas σ_0 is the initial static

$$
\begin{array}{lcl}
\mathcal{E}[\![\text{lift } e]\!]\rho & = & \mathcal{E}[\![e]\!]\rho \\
\mathcal{E}[\![\lambda^D x.e]\!]\rho & = & \eta(\lambda y.\text{withstore } \sigma_0\ (\mathcal{E}[\![e]\!]\rho[y/x])) \\
\mathcal{E}[\![e_1 @^D e_2]\!]\rho & = & \mathcal{E}[\![e_1]\!]\rho \Rightarrow \lambda y_1.\mathcal{E}[\![e_2]\!]\rho \Rightarrow \lambda y_2.\text{withstore } \sigma_\perp\ (y_1 @ y_2) \\
\mathcal{E}[\![\text{fix}^D\ x.e]\!]\rho & = & \text{fixM } \lambda y.\mathcal{E}[\![e]\!]\rho[y/x] \\
\mathcal{E}[\![\text{let}^D\ x = e_1 \text{ in } e_2]\!]\rho & = & \mathcal{E}[\![e_1]\!]\rho \Rightarrow \lambda y_1.\mathcal{E}[\![e_2]\!]\rho[y/x] \\
\mathcal{E}[\![\text{if0}^D\ e_1\ e_2\ e_3]\!]\rho & = & \mathcal{E}[\![e_1]\!]\rho \Rightarrow \lambda y_1.\text{if } (y_1 = 0)(\mathcal{E}[\![e_2]\!]\rho)(\mathcal{E}[\![e_3]\!]\rho) \\
\mathcal{E}[\![\text{ref}^D\ e]\!]\rho & = & \mathcal{E}[\![e]\!]\rho \Rightarrow (\lambda y.\text{alloc}^D \ \Rightarrow (\lambda p.\text{assign}^D\ p\ y)) \\
\mathcal{E}[\![!^D e]\!]\rho & = & \mathcal{E}[\![e]\!]\rho \Rightarrow (\lambda p.\text{fetch}^D\ p) \\
\mathcal{E}[\![e_1 :=^D e_2]\!]\rho & = & \mathcal{E}[\![e_1]\!]\rho \Rightarrow (\lambda p.\mathcal{E}[\![e_2]\!]\rho \Rightarrow (\lambda v.\text{assign}^D\ p\ v)) \\
\mathcal{E}[\![\text{succ}^D\ e]\!]\rho & = & \mathcal{E}[\![e]\!]\rho \Rightarrow \lambda y.\eta(y+1)
\end{array}
$$

Figure 7: Oblivious Interpretation of the Dynamic Constructs

state of a local computation inside of dynamic abstractions, the store $\sigma_\perp$ is always ignored. In consequence their binding times are different: σ_0 is static and $\sigma_\perp$ is dynamic.

The special operations now operate either on the static store σ or on the dynamic store δ depending on the annotations:

$$
\begin{array}{lcl}
\text{alloc } \ell\ y & = & \lambda k\delta\sigma.k(\text{up}(\alpha,\delta,\sigma[\alpha \mapsto y])) \text{ where } \sigma\alpha = \text{unused} \wedge \alpha \sim \ell \\
\text{assign } \alpha\ y & = & \lambda k\delta\sigma.k(\text{up}(y,\delta,\sigma[\alpha \mapsto y])) \\
\text{fetch } \alpha & = & \lambda k\delta\sigma.k(\text{up}(\sigma\alpha,\delta,\sigma)) \\
\text{alloc}^D\ \ell\ y & = & \lambda k\delta\sigma.k(\text{up}(\alpha,\delta[\alpha \mapsto y],\sigma)) \text{ where } \delta\alpha = \text{unused} \wedge \alpha \sim \ell \\
\text{assign}^D\ \alpha\ y & = & \lambda k\delta\sigma.k(\text{up}(y,\delta[\alpha \mapsto y],\sigma)) \\
\text{fetch}^D\ \alpha & = & \lambda k\delta\sigma.k(\text{up}(\delta\alpha,\delta,\sigma))
\end{array}
$$

We aim at interpreting $\mathcal{E}$ as a translation from annotated λ^{ML} terms to annotated λ terms. The translation maps dynamic abstractions and applications to dynamic abstractions and applications; it expands and annotates the monadic operators appropriately. For example, the translation $\mathcal{T}_{2s}[\![_]\!]$ maps $\lambda^D x.e$ to

$$\eta(\lambda^D x.\text{withstore } \sigma_0\ \mathcal{T}_{2s}[\![e]\!])$$

which expands to (ignoring the lifts and strictifications)

$$\lambda k\delta\sigma.k((\lambda^D x.\lambda^D k\delta\sigma'.\mathcal{T}_{2s}[\![e]\!](\lambda(v,\delta,_).k @^D(v,\delta,\sigma'))\delta\sigma_0),\delta,\sigma)$$

where we have added the appropriate annotations. It is important for binding-time reasons that the translated term starts a new static store inside of the dynamic abstraction. Likewise, the result of $\mathcal{T}_{2s}[\![e_1 @^D e_2]\!]$ is

$$\mathcal{T}_{2s}[\![e_1]\!] \Rightarrow \lambda y_1.\mathcal{T}_{2s}[\![e_2]\!] \Rightarrow \lambda y_2.\text{withstore } \sigma_\perp\ (y_1 @ y_2)$$

which expands to (after some simplification)

$$\lambda k\delta\sigma.\mathcal{T}_{2s}[\![e_1]\!](\lambda(y_1,\delta_1,\sigma_1).\mathcal{T}_{2s}[\![e_2]\!](\lambda(y_2,\delta_2,\sigma_2).y_1 @^D y_2(\lambda^D(y_3,\delta_3,\sigma_3).k(y_3,\delta_3,\sigma_2))\delta_2\sigma_\perp)\delta_1\sigma_1)\delta\sigma$$

Here it is important that the dynamic application (of a dynamic function) gets a dummy store $\sigma_\perp$ as its static store parameter in place of the real static store. The static store σ_2 is only passed to the continuation of the dynamic application.

6 Reaching Definitions

Mutable variables often keep values of different binding times during their life time. This has a significant impact on the binding times of the whole program. Therefore it makes sense to split

```
fn n => fn d =>
 let val power =
   (fn n => fn x =>
     let fun pow () = if (!n)=0
                      then (n := 1)
                      else (n := !n - 1; pow (); n := !n * x)
     in (pow (); !n)
     end)
in (power (ref n)) d
end
```

Figure 8: Power Example

```
fn n => fn d =>
  let val power =
   (fn n1 => fn n2 => fn x =>
      let fun pow () = if (!n1)=0
                       then (n2 := 1)
                       else (n1 := !n1 - 1; pow(); n2 := !n2 * x)
      in (pow (); !n2)
      end)
  in  ((power (ref n)) (ref n)) d
  end
```

Figure 9: Binding-Time Improved Power Example

variables if they have non-overlapping life ranges. The following example (Fig. 8) demonstrates the effect with an imperative definition of the power function. It uses the variable n both as a (static) counter for the exponent and to accumulate the (dynamic) result.

The dynamic d forces x to be dynamic which in turn forces the inner n dynamic. Hence, the specializer produces code for everything involving the variable n.

However, it is possible to split n into two variables n1 and n2. One—n1—is the counter and the other—n2—contains the result. This split does not affect the semantics, because the variants of n will never float together. We have developed a global reaching-definitions analysis that identifies the non-overlapping life ranges of a variable. A subsequent program transformation splits the variables according to the result of the analysis. Fig. 9 shows the transformed program and Fig. 10 the residual program for n = 3.

7 Further Issues

7.1 Dynamic Side Effects

The specializer should not reorder, discard, nor duplicate side effects. We do not change the specializer to this respect but rely on a preprocessing step that inserts dynamic let-expressions whenever a dynamic value is put in a static context [10, 8]. For example, if $\mathrm{top}(\beta) = D$ transform

$$
\begin{array}{lcl}
\lambda^S(x:\beta).e & \Rightarrow & \lambda^S x.(\mathrm{let}^D\ x = x \text{ in } e) \\
\mathrm{ref}^S\ (e:\beta) & \Rightarrow & \mathrm{let}^D\ x^\diamond = e \text{ in } \mathrm{ref}^S\ x^\diamond \\
e_1 :=^S (e_2:\beta) & \Rightarrow & e_1 :=^S (\mathrm{let}^D\ x^\diamond = e_2 \text{ in } x^\diamond)
\end{array}
$$

Performing the same transformation also for the dynamic arguments of static data constructors

```
fn d =>
  let val x1 = ref 3
  in  x1 := 1; x1 := !x1 * d; x1 := !x1 * d; x1 := !x1 *d; !x1
  end
```

Figure 10: Residual Expression for Power Example

ensures correct handling of partially static data and variable splitting. For example, our implementation specializes a static list of dynamic references into many different variables, thus removing an inherited limited of other specializers [57].

7.2 Managing the Static Store

Specialization executes program parts out of order: in order to specialize a conditional expression, both branches must be specialized in some fixed order, say, the then-branch first. As long as we stay in the pure world this is not a problem, but in the presence of static assignment it becomes one. Before we specialize the else-branch of a conditional we have to reset the static store to its state before entering the then-branch. The naïve solution for a language such as Pascal or C saves the whole store before entering the then-branch and restores it before entering the else-branch [3]. In the context of a higher-order language this implies saving heap images. Or we would have to come up with some clever analysis, e.g., a higher-order extension of configuration analysis [13].

We solve this problem by using the continuation/store monad, which provides special store operators. In the same way as continuation-based specializer can be written in direct style with control operators [49], we can implement continuation/store-based specializers in direct style by using a direct implementation of curstore and withstore .

The solution consists in adapting the concept of first-class stores [59, 45, 25] to our setting. A language supporting first-class stores provides operations to reify the current store and to reflect a store (to become current). Thus it is possible to "save" the current store as a data object and return to it later. This is precisely the functionality we need for implementing curstore and withstore in the continuation/store monad.

7.3 Combinators for PGGs

Monad-based partial evaluation is also amenable to the construction of PGGs generalizing our previous development for continuation-based partial evaluation [77]. After changing the type of the environment from Var $\rightarrow$ Val to Var $\rightarrow$ M Val and converting the syntax to higher-order abstract syntax we obtain the combinator definitions in Fig. 11. Due to the change of the type of the environment, generating extensions need not encode variables, but they can employ meta-variables for variables.

In the transition to higher-order abstract syntax we replace the binding constructs $\lambda x.e$, fix $x.e$, and let $x = e$ in e by *lamf*, *fixf*, and *let e in f*. In each case f is a function that maps a variable name to an expression.

7.4 Dynamically-Typed Languages

Our framework for BTA applies directly to dynamically typed languages such as Scheme [44]. Instead of starting with the simple type system for the ML subset, we start with a dynamic type system [41]. Another problem of Scheme is the presence of second class references (mutable variables). A prepass which introduces an explicit box for every such variable solves this problem.

$$
\begin{aligned}
0 &= \eta(0) \\
\text{succ } e_1 &= e_1 \Rightarrow (\lambda y.\eta(y+1)) \\
x &= x \\
\mathit{lam} f &= \eta(\lambda y.f(\eta(y))) \\
e_1 @ e_2 &= e_1 \Rightarrow (\lambda f.e_2 \Rightarrow f) \\
\mathit{fix} f &= \text{fixM } \lambda y.f(\eta(y)) \\
\mathit{let}\ e\ \mathit{in}\ f &= e \Rightarrow (\lambda y.f(\eta(y))) \\
\text{if0 } e_1\ e_2\ e_3 &= e_1 \Rightarrow (\lambda x.\text{if } (x=0) e_2 e_3) \\
\text{ref}^{\ell}\ e &= e \Rightarrow (\lambda y.\text{alloc } \ell\ y) \\
!e &= e \Rightarrow (\lambda p.\text{fetch } p) \\
e_1 := e_2 &= e_1 \Rightarrow (\lambda p.e_2 \Rightarrow (\lambda v.\text{assign } p\ v)) \\
\\
\text{lift } e &= e \Rightarrow \lambda y.\eta(\mathit{quote}(y)) \\
\text{succ}^D\ e &= e \Rightarrow \lambda y.\eta(\underline{\text{succ}}\ y) \\
\mathit{lam}^D f &= \text{reset } (f(\eta(x^\diamond))) \Rightarrow \lambda y.\eta(\underline{\lambda} x^\diamond.y) \\
e_1 @^D e_2 &= e_1 \Rightarrow \lambda y_1.e_2 \Rightarrow \lambda y_2.\eta(y_1 \underline{@} y_2) \\
\mathit{fix}^D f &= \text{reset } (f(\eta(x^\diamond))) \Rightarrow \lambda y.\eta(\underline{\text{fix}}\ x^\diamond.y) \\
\mathit{let}^D\ e\ \mathit{in}\ f &= \text{shift } \lambda j.e \Rightarrow \lambda y_1. \\
&\qquad \text{reset } (f(\eta(x^\diamond)) \Rightarrow j) \Rightarrow \lambda y_2. \\
&\qquad \eta(\underline{\text{let}}\ x^\diamond = y_1\ \underline{\text{in}}\ y_2) \\
\text{if0}^D\ e_1\ e_2\ e_3 &= \text{shift } \lambda j.e_1 \Rightarrow \lambda y_1. \\
&\qquad \text{curstore} \Rightarrow \lambda s.\text{reset } (e_2 \Rightarrow j) \Rightarrow \lambda y_2. \\
&\qquad \text{withstore } s \text{ (reset } (e_3 \Rightarrow j)) \Rightarrow \lambda y_3. \\
&\qquad \eta(\underline{\text{if0}}\ y_1\ y_2\ y_3) \\
\text{ref}^D\ e &= e \Rightarrow \lambda y.\eta(\underline{\text{ref}}\ y) \\
!^D e &= e \Rightarrow \lambda y.\eta(\underline{!} y) \\
e_1 :=^D e_2 &= e_1 \Rightarrow \lambda y_1.e_2 \Rightarrow \lambda y_2.\eta(y_1 \underline{:=} y_2)
\end{aligned}
$$

Figure 11: Monadic Combinators for PGGs

8 Related Work

Specialization of programs with side effects has a long history. Essentially all of the early works consider online specialization for imperative languages. For example Futamura [35] and Ershov and his group [29, 30, 12, 31] consider fragments of Algol. Building on work by Ershov's group, Meyer [56] defines and proves correct an online specializer for Pascal that performs side effects at specialization time. Marquard and Steensgaard [55] describe an online partial evaluator for a simple object-oriented imperative language. On the side of functional languages, the REDFUN group [6, 39] developed specializers for impure Lisp. However, their treatment of side effects is fairly ad-hoc and they do not provide formal correctness arguments. Ørbæk's POPE [69] specializes Scheme with mutable variables, but always residualizes operations that affect variables. Asai [4] describe a specializer for a subset of Scheme which handles side-effects using pre-actions. There are also version of the FUSE family of specializers [83] that deal with assignments. Online specializers suffer from the well-known efficiency and termination problems that we wish to avoid.

The first offline specializer for an imperative flow-chart language is flow-chart mix [46]. Today there are partial evaluators for realistic languages, for example Andersen's C-mix for a substantial subset of ANSI C [3], F-spec for a subset of Fortran 77 [5], and M2MIX for Modula-2 [13]. All of them perform some side effects statically, but all of them deal with first-order languages and some do not include pointers (flow-chart mix and F-spec). In contrast, we specialize a higher-order language with first-class mutable references.

Closest to our work are the specializers of Bondorf and Danvy [10, 8] for first-order and higher-order recursive equations with global variables. Neither performs side effects at specialization, but rather defers them to run time. SML-mix [7] and Pell-Mell [54] employ the same techniques for partial evaluation for SML. Pell-Mell employs a similar strategy of wrapping dynamic values in dynamic let-expressions to obtain "lightweight symbolic values."

The first successful use of monadic expansion in program specialization is Consel and Danvy's improvement of static data flow by CPS transformation [15]. CPS transformation is just monadic expansion for the continuation monad. The inefficiency of residual programs in CPS lead to Danvy developing an inverse transformation from continuation-passing style back to direct style [20, 21].

The second step, writing the specializer in monadic style was also first carried out for the continuation monad: Bondorf [9] writes the specializer using continuations and liberalizes the BTA.

The final step, writing the specializer in direct style with special monadic operators was also first proposed for the continuation monad: Lawall and Danvy [49] reexpress Bondorf's specializer in direct style using control operators (special monadic operators of the continuation monad). The implementation of the control operators relies on Filinski's work on representing monads [33]. He shows that any monad whose operations have definitions by lambda terms expressed in purely functional terms is representable in a call-by-value language with call/cc and a single storage cell.

Similarly, Moura, Consel, and Lawall [61] suggest an approach to static analysis of imperative programs by transforming them to a sophisticated variant of store-passing style (and to static single assignment form [19]) and apply analysis techniques developed for pure functional programs. In our terminology, they apply monadic expansion before they analyze a program. As an application they suggest to specialize the expanded program. They do not consider the remaining steps, neither writing the specializer in store-passing style, nor the use of store operators.

Our BTA is inspired by binding-time specifications using non-standard and annotated type systems [37, 40, 27, 36]. The supporting analyses are based on work on effect systems [53] by Jouvelot and others [75, 76].

Furthermore, there are connections to Riecke's work [72], where he considers a language VPCF+S which is very similar to our λ^{ML}. He proposes effect delimiters that confine the scope of an effect (for example on the store) to the delimited expression. In our work, dynamic abstractions play a similar rôle as delimiters of static effects.

The generic construction of monadic PGGs generalizes the second author's construction of continuation-based PGGs [77].

Reaching definitions is a standard analysis in conventional compilers [2]. Our analysis generalizes reaching definitions to the higher-order case and first-class references.

9 Conclusion

We have developed an offline specializer for an ML-like core language. Our specializer is amenable of specializing higher-order programs written in an object-oriented programming style. We have applied the specializer to a good number of classical object-oriented textbook examples and have observed surprisingly good specialization results.

Our novel monadic specialization framework generalizes all existing approaches to partial evaluation for functional languages, including continuation-based partial evaluation. The particular instance we consider is continuation/store-based partial evaluation. Other monads are interesting for specializers, for example an exception monad facilitates specializing programs with exceptions or a non-determinism monad gives rise to a specializer for a non-deterministic lambda calculus with choice operators.

Monads not only provide a clean way to integrate assignments and I/O into pure functional languages (cf. Peyton Jones and Wadler's paper "Imperative Functional Programming" [64]), but that they also provide a general framework for writing specializers with arbitrary monadic effects.

We have implemented our specializer in direct style using control operators (shift/reset) and store operators (curstore/withstore). We use Filinski's direct-style implementation [33] for the control operators and ideas from the implementation of first-class stores [45, 59] to implement the store operators.

Our specializer relies on program analyses, such as reaching definitions, effect analysis and BTA. We have constructed all analyses in a typed-based program analysis framework. The correctness proof of our BTA is modular. It relates the result of the analysis with the result of a BTA of the program after monadic expansion.

Although presented for a simple core language, the techniques scale up to full Scheme or Standard ML. It is straightforward to include partially static data, to construct program generator generators (cogens), and to extend the specializer to multiple levels of binding times.

10 Further Work

Our current BTA yields surprisingly good results, but it is sometimes too conservative. We are currently investigating ways to liberalize the binding-time constraints by using ideas from linear and affine type systems [79, 78] and Reynolds' work on syntactic control of interference [70, 71]. On a more general scale we would like to investigate the impact of polymorphism on the binding-time and effect analysis.

The first author [26] proposes a partial CPS transformation that achieves the same improvement of static data flow as continuation-based partial evaluation using a straightforward specializer. It may be possible to extend this result to the present monadic framework.

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., 1985.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, Dept. of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 København Ø, May 1994.

[4] Kenichi Asai. Toward partial evaluation of side-effecting scheme. Talk at the Dagstuhl Seminar on Partial Evaluation, February 1996.

[5] Romana Baier, Robert Glück, and Robert Zöchling. Partial evaluation of numerical programs in Fortran. In Sestoft and Søndergaard [74], pages 119–132.

[6] L. Beckman, A. Haraldsson, Ö. Oskarsson, and E. Sandewall. A partial evaluator, and its use as a programming tool. *Artificial Intelligence*, 7(4):319–357, 1976.

[7] Lars Birkedal and Morten Welinder. Partial evaluation of Standard ML. Rapport 93/22, DIKU, University of Copenhagen, 1993.

[8] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Programming*, 17:3–34, 1991.

[9] 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.

[10] 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.

[11] Anders Bondorf and Jesper Jørgensen. Efficient analysis for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, July 1993.

[12] Mikhail A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.

[13] Mikhail A. Bulyonkov and Dmitrij V. Kochetov. Practical aspects of specialization of Algol-like programs. In Danvy et al. [24], pages 17–32.

[14] Charles Consel. A tour of Schism. In David Schmidt, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '93*, pages 134–154, Copenhagen, Denmark, June 1993. ACM Press.

[15] Charles Consel and Olivier Danvy. For a better support of static data flow. In Hughes [43], pages 496–519. LNCS 523.

[16] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In POPL1993 [66], pages 493–501.

[17] 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. [24], pages 54–72.

[18] 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.

[19] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control flow graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[20] Olivier Danvy. Back to direct style. In Bernd Krieg-Brückner, editor, *Proc. 4th European Symposium on Programming '92*, pages 130–150, Rennes, France, February 1992. Springer-Verlag. LNCS 582.

[21] Olivier Danvy. Back to direct style. *Science of Programming*, 22:183–195, 1994.

[22] Olivier Danvy and Andrzej Filinski. Abstracting control. In LFP 1990 [50], pages 151–160.

[23] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2:361–391, 1992.

[24] Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, Dagstuhl, Germany, February 1996. Springer Verlag, Heidelberg.

[25] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *Proceedings of the Eighteenth ACM Symposium on Theory of Computing*, pages 109–121, May 1986.

[26] Dirk Dussart and Eddy Bevers. CPS transformation after binding-time analysis (extended abstract). In *Proceedings of the 7th Nordic Workshop on Programming Theory*, pages 112–126, Göteborg, Sweden, November 1995.

[27] Dirk Dussart, Fritz Henglein, and Christian Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In Alan Mycroft, editor, *Proc. International Static Analysis Symposium, SAS'95*, pages 118–136, Glasgow, Scotland, September 1995. Springer-Verlag. LNCS 983.

[28] Dirk Dussart and Peter Thiemann. Imperative functional partial evaluation. Berichte des Wilhelm-Schickard-Instituts WSI-96-XX, Universität Tübingen, July 1996.

[29] Andrei P. Ershov. On the essence of compilation. In Erich J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 391–420. North-Holland, 1978.

[30] Andrei P. Ershov. On mixed computation: Informatl account of the strict and polyvariant computational schemes. In Manfred Broy, editor, *Control Flow and Data Flow: Consepts of Distributed Programming*, pages 107–120. Springer-Verlag, 1984.

[31] Andrei P. Ershov and B. N. Ostrovsky. Controlled mixed computation and its application to systematic development of language-oriented parsers. In Lambert G. L. T. Meertens, editor, *Program Specification and Transformation, Proc. IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation*, pages 31–48. North Holland, 1987.

[32] David Espinosa. Building interpreters by transforming stratified monads. ftp://altdorf.ai.mit.edu/pub/dae, June 1994.

[33] Andrzej Filinski. Representing monads. In POPL1994 [67], pages 446–457.

[34] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press and McGraw-Hill, 1992.

[35] Yoshihiko Futamura. Partial evaluation of computation process — an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[36] Robert Glück and Jesper Jørgensen. Fast multi-level binding-time analysis for multiple program specialization. In *PSI-96: Andrei Ershov Second International Memorial Conference, Perspectives of System Informatics*, Novosibirsk, Russia, June 1996.

[37] Carsten K. Gomard. Partial type inference for untyped functional programs. In LFP 1990 [50], pages 282–287.

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

[39] Anders Haraldsson. *A Program Manipulation System Based on Partial Evaluation*. PhD thesis, Linköping University, Sweden, 1977. Linköping Studies in Science and Technology Dissertations 14.

[40] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In Hughes [43], pages 448–472. LNCS 523.

[41] Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Programming*, 22:197–230, 1994.

[42] Paul Hudak and Neil D. Jones, editors. *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '91*, New Haven, CT, June 1991. ACM. SIGPLAN Notices 26(9).

[43] John Hughes, editor. *Functional Programming Languages and Computer Architecture*, Cambridge, MA, 1991. Springer-Verlag. LNCS 523.

[44] Institute of Electrical and Electronic Engineers, Inc. IEEE standard for the Scheme programming language. IEEE Std 1178-1990, New York, NY, 1991.

[45] G. F. Johnson and Dominic Duggan. Stores and partial continuations as first-class objects in a language and its environment. In POPL1988 [65], pages 158–168.

[46] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[47] 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.

[48] John Launchbury and Simon L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 1995. to appear.

[49] 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.

[50] *Proc. 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, 1990. ACM Press.

[51] Sheng Liang and Paul Hudak. Modular denotational semantics for compiler construction. In Hanne Riis Nielson, editor, *Proc. 6th European Symposium on Programming*, page ??, Linköping, Sweden, April 1994. Springer-Verlag. LNCS.

[52] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In POPL1995 [68], pages 333–343.

[53] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In POPL1988 [65], pages 47–57.

[54] Karoline Malmkjær, Olivier Danvy, and Nevin Heintze. ML partial evaluation using set-based analysis. In *Record of the ACM-SIGPLAN Workshop on ML and its Applications*, number 2265 in INRIA Research Report, pages 112–119, BP 105, 78153 Le Chesnay Cedex, France, June 1994.

[55] Morten Marquard and Bjarne Steensgaard. Partial evaluation of an object-oriented imperative language. Master's thesis, Department of Computer Science, University of Copenhagen, Denmark, April 1992.

[56] Uwe Meyer. Techniques for partial evaluation of imperative languages. In Hudak and Jones [42], pages 94–105. SIGPLAN Notices 26(9).

[57] Torben Æ. Mogensen. Evolution of partial evaluators: Removing inherited limits. In Danvy et al. [24], pages 303–321.

[58] Eugenio Moggi. Computational lambda-calculus and monads. In *Proc. of the 4rd Annual Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, CA, June 1989. IEEE Computer Society Press.

[59] J. Gregory Morrisett. Generalizing first-class stores. In Paul Hudak, editor, *SIPL '93, ACM SIGPLAN Workshop on State in Programming Languages*, number YALEU/DCS/RR-968, pages 73–87, New Haven, CT, June 1993. Copenhagen, Denmark.

[60] Peter D. Mosses. *Denotational Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 11. Elsevier Science Publishers, Amsterdam, 1990.

[61] Bàrbara Moura, Charles Consel, and Julia Lawall. Bridging the gap between functional and imperative languages. Publication interne 1027, Irisa, Rennes, France, July 1996.

[62] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*. Cambridge University Press, 1992.

[63] Jens Palsberg. Correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):347–364, July 1993.

[64] Simon L. Peyton Jones and Philip L. Wadler. Imperative functional programming. In POPL1993 [66], pages 71–84.

[65] *Proc. 15th Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988.

[66] *Proc. 20th Annual ACM Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 1993. ACM Press.

[67] *Proc. 21st Annual ACM Symposium on Principles of Programming Languages*, Portland, OG, January 1994. ACM Press.

[68] *Proc. 22nd Annual ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 1995. ACM Press.

[69] Peter Ørbæk. POPE: An on-line partial evaluator. ftp://ftp.daimi.aau.dk/pub/empl/poe/pope.ps.gz, June 1994.

[70] John C. Reynolds. Syntactic control of interference. In *Proc. 5th Annual ACM Symposium on Principles of Programming Languages*, pages 39–46, Tucson, Arizona, January 1978. ACM Press.

[71] John C. Reynolds. Syntactic control of interference, part 2. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simona Ronchi Della Rocca, editors, *Automata, Languages and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 704–722, Stresa, Italy, 1989. Springer-Verlag, Berlin.

[72] 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.

[73] David A. Schmidt. *Denotational Semantics, A Methodology for Software Development*. Allyn and Bacon, Inc, Massachusetts, 1986.

[74] Peter Sestoft and Harald Søndergaard, editors. *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM '94*, Orlando, Fla., June 1994. ACM.

[75] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–272, July 1992.

[76] Yan-Mei Tang and Jean-Pierre Talpin. Effect systems with subtyping. Technical report, CRI, Ecole des Mines de Paris, 1993.

[77] Peter Thiemann. Cogen in six lines. In R. Kent Dybvig, editor, *Proc. International Conference on Functional Programming 1996*, pages 180–189, Philadelphia, PA, May 1996. ACM Press, New York.

[78] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In Simon Peyton Jones, editor, *Proc. Functional Programming Languages and Computer Architecture 1995*, pages 1–11, La Jolla, CA, June 1995. ACM Press, New York.

[79] Philip Wadler. Is there a use for linear logic? In Hudak and Jones [42], pages 255–273. SIGPLAN Notices 26(9).

[80] Philip L. Wadler. Comprehending monads. In LFP 1990 [50], pages 61–78.

[81] 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.

[82] Mitchell Wand. Specifying the correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):365–387, July 1993.

[83] Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic online partial evaluation. In Hughes [43], pages 165–191. LNCS 523.