Type Specialisation for Imperative Languages

Dirk Dussart* John Hughes† Peter Thiemann‡

Abstract

We extend type specialisation to a computational lambda calculus with first-class references. The resulting specialiser has been used to specialise a self-interpreter for this typed computational lambda calculus optimally. Furthermore, this specialiser can perform operations on references at specialisation time, when possible.

Keywords: program transformation, specialisation, type systems, monads.

1 Introduction

Partial evaluation is an important program specialisation technique [18]. Partial evaluation is a program transformation, which transforms a program and parts of its input (the *static* part) into a new program. The partial evaluator performs the operations for which enough data is available and reconstructs the rest. A partial evaluator operates on both *values and symbolic values* (code or values or a combination of code and values), whereas an interpreter operates *only* on values. Partial evaluators appear in two variants, *online* and *offline*. In online partial evaluators all specialisation decisions are taken by inspecting the symbolic values, whereas in the offline variants all specialisation decisions are taken before actually running the specialiser. The decisions are communicated to the specialiser by means of two-level annotated programs [27]. These annotations can be introduced by hand or by means of a *binding-time analysis*. The specialiser, in this case, is an interpreter for two-level programs.

Type specialisation [17] is an extension of offline partial evaluation, where the specialiser is defined by a collection of non-standard typing rules. The rules specify specialisation judgements, $\Gamma \mid\!\!- e : \tau \hookrightarrow e' : \tau'$, which denote transformations of expressions e of type τ into expressions e' of type τ'. e' is the code part or *dynamic* part of the result of applying the transformation, and τ' plays the same rôle as the symbolic values in partial evaluation, and may contain values. The static parts can be used in static computations and the *dynamic* parts are treated as black-boxes. In this setting, specialisation corresponds to building a proof of a specialisation judgement. The advantage over partial evaluation is that symbolic values, the residual types, are propagated via type unification[1]. The combination of a rich language of residual types and the superior value propagation strategy, type unification, has enabled type specialisation to achieve optimal specialisation of a self-interpreter for a typed lambda calculus [17], a feat that has not been achieved with standard partial evaluation. Optimal specialisation in this context means that specialising the interpreter with respect to some term yields the same term up to α-conversion. The problem here is the presence of a universal type in the self-interpreter. Standard partial evaluation is not able to get rid of the tagging and untagging operations of the universal type [19].

This work serves two purposes. First, we show that type specialisation is *modular* with respect to the addition of new type constructors (here: references and computations). Second, we show that type specialisation is not restricted to purely functional languages, but that it also applies to specialising operations with computational effects, such as performing operations on references at specialisation time for a language like ML. More formally, we extend type specialisation to an extension of Moggi's computational metalanguage Λ_{ml} with first-class references. We demonstrate the usefulness of this extension by instantiating it with the store monad. The result is a type specialiser for Λ^{σ}_{ml}, that is Λ_{ml} extended with ML-style operations on references. A distinguishing feature of our specialiser is the fact that it can perform some operations on references statically. This specializer achieves optimal specialisation for a self-interpreter for Λ^{σ}_{ml} and it also yields satisfactory results on other problems (to be described below).

Overview For the reader unfamiliar with type specialisation, we start with a short type specialisation primer. Further information can be found in the original paper on type specialisation [17]. Then we introduce the language Λ^{σ}_{ml}, an extension of Moggi's computational lambda calculus with

*Department of Computer Science, K.U.Leuven, Celestijnenlaan 200A, B-3001 Leuven, Belgium. Dirk.Dussart@cs.kuleuven.ac.be.
Supported by the Belgian National Fund for Scientific Research (N.F.W.O.). This work was done while visiting Chalmers.
†Chalmers Tekniska Högskola, S-41296 Göteborg, Sweden. rjmh@cs.chalmers.se.

‡Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Sand 13, D-72076 Tübingen, Germany. thiemann@informatik.uni-tuebingen.de. This work was done while visiting Chalmers.

In: Proceedings of the International Conference on Functional Programming, ICFP'97, Amsterdam, The Netherlands, June 1997.

[1] More information can be found in Section 2

first-class references. As a first step towards the full specialiser, we state the specialisation rules for the treatment of dynamic reference operations. We apply this specialiser to a self-interpreter for Λ^{σ}_{ml} and achieve optimal specialisation. In a second step we add specialisation rules for static references and apply this specialiser to an interpreter for a lazy functional language that implements laziness using updatable closures. Next, we discuss the addition of store prompts and state threads and state the new postprocessing rules. Finally, we assess our work with respect to partial evaluation, consider related work, and conclude.

2 Type Specialisation Primer

Type specialisation is a form of program specialisation in which types play a central rôle. Rather than saying that one term specialises to another term, we say that a source term *and type* specialise to a residual term *and type*: $e : \tau \hookrightarrow e' : \tau'$. Source types carry information to control specialisation, in particular binding-times: most type formers come in static and dynamic variants, and we mark the dynamic ones by underlining as usual. Residual types are sufficiently refined to carry all static information, which means that static values need not appear at all in residual terms. For example, static integer constants are specialised to dummy values, $42 : \mathbf{int} \hookrightarrow \bullet : 42$, where the residual type records which integer the dummy value represents. The residual type information is enough to specialise expressions that use static values appropriately: for example, the **lift** operation, which converts a static integer to a dynamic one, specialises to the constant value given by the residual type of its argument: $\mathbf{lift}\ (2+3) : \underline{\mathbf{int}} \hookrightarrow 5 : \mathbf{int}$.

Well-typedness of the source term can be checked once and for all before specialisation, but well-typedness of the residual term can be checked only as it is constructed, during specialisation. Thus the specialiser embodies a residual type-checker, and it is the unifications in this type-checker that propagate static information during specialisation. For example, specialising

$$\underline{\mathbf{let}}\ f = \underline{\lambda} x.\mathbf{lift}\ (x+1)\ \underline{\mathbf{in}}\ f \mathbin{\underline{@}} 3 : \underline{\mathbf{int}}$$

(where $\underline{@}$ is dynamic function application) yields

$$\mathbf{let}\ f = \lambda x.4\ \mathbf{in}\ f\ \bullet : \mathbf{int}$$

where unification assigns x the residual type 3, and this is sufficient information to perform the static addition and specialise the **lift** to 4.

Type specialisation can propagate static information more effectively than partial evaluation can. Modifying the example slightly, we can specialise

$$\underline{\mathbf{let}}\ f = \underline{\lambda} x.x+1\ \underline{\mathbf{in}}\ \mathbf{lift}\ (f \mathbin{\underline{@}} 3) : \underline{\mathbf{int}}$$

to

$$\mathbf{let}\ f = \lambda x.\bullet\ \ \mathbf{in}\ 4 : \mathbf{int}$$

since unification assigns x the residual type 3, which enables the static addition to be performed, allowing f to be assigned residual type $3 \rightarrow 4$, which enables the **lift** to be specialised to a constant. This can be done *even though f is a dynamic function, whose calls may not be unfolded.* Conventional partial evaluators do not allow dynamic functions to return static results, since they derive static values by reducing expressions to their normal form, and dynamic function applications must not be reduced.

Type specialisation is inherently *monovariant*; if we modify our example to

$$\underline{\mathbf{let}}\ f = \underline{\lambda} x.\mathbf{lift}\ (x+1)\ \underline{\mathbf{in}}\ f \mathbin{\underline{@}} 3 \mathbin{\underline{+}} f \mathbin{\underline{@}} 4 : \underline{\mathbf{int}}$$

then it can no longer be specialised, because f would need to be assigned both $3 \rightarrow \mathbf{int}$ and $4 \rightarrow \mathbf{int}$ as residual types. But polyvariance is easily added via a new source type $\mathbf{poly}\ \tau$, with constructor **poly** and selector **spec**. An expression $\mathbf{poly}\ e$ specialises to a tuple of specialisations of e, while $\mathbf{spec}\ e$ specialises to a selection from the tuple. If we rewrite our example as

$$\underline{\mathbf{let}}\ f = \mathbf{poly}\ \underline{\lambda} x.\mathbf{lift}\ (x+1)\ \underline{\mathbf{in}}\ \mathbf{spec}\ f \mathbin{\underline{@}} 3 \mathbin{\underline{+}} \mathbf{spec}\ f \mathbin{\underline{@}} 4 : \underline{\mathbf{int}}$$

then it specialises to

$$\mathbf{let}\ f = (\lambda x.4, \lambda x.5)\ \mathbf{in}\ \pi_1\ f\ \ \bullet + \pi_2\ f\ \bullet : \mathbf{int}$$

where f has residual type $(3 \rightarrow \mathbf{int}, 4 \rightarrow \mathbf{int})$. The residual type information is enough to enable **spec** to choose the right version at each call, and conversely the type information at each **spec** is enough to enable **poly** to decide which versions to create.

Alternatively, we could change f in the example into a *static function*:

$$\underline{\mathbf{let}}\ f = \lambda x.\mathbf{lift}\ (x+1)\ \underline{\mathbf{in}}\ f\ 3 \mathbin{\underline{+}} f\ 4 : \underline{\mathbf{int}}$$

(note that the λ and the applications are no longer underlined). Static functions are unfolded during specialisation, so in this example each call of f is replaced by a specialisation of its body, and of course the two specialisations can be different. In order to unfold static function applications, the specialiser represents a static function as a closure. The static parts of the closure (bound variable name, body, and residual types of the free variables) make up the residual type of a static λ, while the dynamic part (a tuple of the free variable values) becomes the residual term. The residual type of a static function says *nothing* about the type of its argument or result, and so such a function can be freely applied to different static arguments at different calls. In this example f has no free variables, and so its residual term is the empty tuple (or dummy value). The result of specialisation is

$$\mathbf{let}\ f = \bullet\ \mathbf{in}\ 4+5 : \mathbf{int}$$

As these examples reveal, static computations leave an undesirable 'residue' of dummy values in the specialised programs. But an optimal specialiser must remove static computations completely. Fortunately, most dummy values can be removed by a post-processing phase we call *void erasure*. Completely static expressions yield residual types with only a single element, for example 3, 4, $3 \rightarrow 4$, or 3×4. All such expressions can safely be replaced by the dummy value $\bullet$, and all such types by **void**. Now using the isomorphisms

$$\mathbf{void} \rightarrow \tau \simeq \tau \qquad \mathbf{void} \times \tau \simeq \tau$$

we can eliminate void components, void parameters, and of course let-bound variables of void type. Void erasure often

$$
\begin{array}{lll}
e & ::= & x \mid n \mid \mathbf{lift}\ e \mid \lambda x.e \mid e\ e \mid \mathbf{let}\ x = e\ \mathbf{in}\ e \mid \underline{\lambda} x.e \mid e \mathbin{\underline{@}} e \mid \underline{\mathbf{let}}\ x = e\ \underline{\mathbf{in}}\ e \mid \mathbf{poly}\ e \mid \mathbf{spec}\ e \quad \text{terms} \\
\tau & ::= & \mathbf{int} \mid \underline{\mathbf{int}} \mid \tau \to \tau \mid \tau \mathbin{\underline{\to}} \tau \mid \mathbf{poly}\ \tau \quad \text{types}
\end{array}
$$

$$\Gamma, x : \tau \vdash x : \tau \qquad \Gamma \vdash n : \mathbf{int}$$

$$\frac{\Gamma \vdash e : \mathbf{int}}{\Gamma \vdash \mathbf{lift}\ e : \underline{\mathbf{int}}} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{poly}\ e : \mathbf{poly}\ \tau} \qquad \frac{\Gamma \vdash e : \mathbf{poly}\ \tau}{\Gamma \vdash \mathbf{spec}\ e : \tau}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \underline{\lambda} x.e : \tau_1 \mathbin{\underline{\to}} \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \mathbin{\underline{\to}} \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 \mathbin{\underline{@}} e_2 : \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \underline{\mathbf{let}}\ x = e_1\ \underline{\mathbf{in}}\ e_2 : \tau_2}$$

Figure 1: The Two-level Source Language

$$\tau' \quad ::= \quad n \mid \mathbf{int} \mid \tau' \to \tau' \mid (\tau' \ldots \tau') \quad \text{residual types}$$

$$\Gamma \vdash n : \mathbf{int} \hookrightarrow \bullet : n \qquad \Gamma, x : \tau \hookrightarrow e : \tau' \vdash x : \tau \hookrightarrow e : \tau' \qquad \frac{\Gamma \vdash e : \mathbf{int} \hookrightarrow e' : n}{\Gamma \vdash \mathbf{lift}\ e : \underline{\mathbf{int}} \hookrightarrow n : \mathbf{int}}$$

$$\frac{\Gamma, x : \sigma \hookrightarrow x' : \sigma' \vdash e : \tau \hookrightarrow e' : \tau'}{\Gamma \vdash \underline{\lambda} x.e : \sigma \mathbin{\underline{\to}} \tau \hookrightarrow \lambda x'.e' : \sigma' \to \tau'}\, x' \text{ fresh} \qquad \frac{\Gamma \vdash e_1 : \sigma \mathbin{\underline{\to}} \tau \hookrightarrow e_1' : \sigma' \to \tau' \quad \Gamma \vdash e_2 : \sigma \hookrightarrow e_2' : \sigma'}{\Gamma \vdash e_1 \mathbin{\underline{@}} e_2 : \tau \hookrightarrow e_1'\ e_2' : \tau'}$$

$$\frac{\Gamma \vdash e : \tau \hookrightarrow e_1 : \tau_1 \quad \cdots \quad \Gamma \vdash e : \tau \hookrightarrow e_n : \tau_n}{\Gamma \vdash \mathbf{poly}\ e : \mathbf{poly}\ \tau \hookrightarrow (e_1 \ldots e_n) : (\tau_1 \ldots \tau_n)} \qquad \frac{\Gamma \vdash e : \mathbf{poly}\ \tau \hookrightarrow e' : (\tau_1 \ldots \tau_n)}{\Gamma \vdash \mathbf{spec}\ e : \tau \hookrightarrow \pi_i\ e' : \tau_i}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \hookrightarrow e_1' : \tau_1' \quad \begin{array}{l}\Gamma, x : \tau_1 \hookrightarrow e_1' : \tau_1' \vdash \\ \qquad e_2 : \tau_2 \hookrightarrow e_2' : \tau_2'\end{array}}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2 \hookrightarrow e_2' : \tau_2'} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \hookrightarrow e_1' : \tau_1' \quad \begin{array}{l}\Gamma, x : \tau_1 \hookrightarrow x' : \tau_1' \vdash \\ \qquad e_2 : \tau_2 \hookrightarrow e_2' : \tau_2'\end{array}}{\Gamma \vdash \underline{\mathbf{let}}\ x = e_1\ \underline{\mathbf{in}}\ e_2 : \tau_2 \hookrightarrow \mathbf{let}\ x' = e_1'\ \mathbf{in}\ e_2' : \tau_2'}\, x' \text{ fresh}$$

Figure 2: Basic Specialisation Rules

simplifies residual programs drastically; for example, the last two specialisations above become just

$$\mathbf{let}\ f = (4, 5)\ \mathbf{in}\ \pi_1\ f + \pi_2\ f : \mathbf{int}$$

and

$$4 + 5 : \mathbf{int}$$

Optimal specialisation of typed interpreters demands that we specialise the *universal type* that such interpreters use to represent values appropriately. For example, an interpreter for λ-calculus with integers might use the type

$$\mathbf{data}\ \mathit{Univ} = \ \mathit{Num}\ \underline{\mathbf{int}} \mid \mathit{Fun}\ (\mathit{Univ} \mathbin{\underline{\to}} \mathit{Univ})$$

to represent values. This is an example of a *static sum type*: the constructors encode the type of interpreted values, and must be known statically. We therefore embed such static constructors into residual types, rather than residual terms. For example

$$\mathit{Num}\ (\mathbf{lift}\ (1 + 2)) : \mathit{Univ} \hookrightarrow 3 : \mathit{Num}\ \mathbf{int}$$

The residual type records the fact that a *Num* constructor was applied, and so it need not appear at all in the residual term. Case expressions over static sums can be simplified to one branch, given the residual type of the inspected expression, and so neither run-time type tags, nor run-time tag checks, need appear in the residual programs. An interpreter for Λ^{σ}_{ml} that can be specialised optimally will be given below.

Of course, partial evaluators can also implement static sum types, and keep track of constructors statically. But to specialise interpreters optimally, we need to be able to return values of a static sum type *from a dynamic function*. For the type specialiser which is deriving static information by type inference, this is no problem. For a partial evaluator which is using reduction, it is impossible.

In Figure 1 we give the syntax and two-level type system for a large fragment of our meta-language. It is essentially a two-level version of the simply-typed λ-calculus. The full language also includes sums and products, which we have omitted here. We allow types to be recursive with no special syntax. The point we wish to emphasise here is that types can be *freely* formed according to the syntax given; there are no 'well-formedness' conditions, such as that the argument and result of a dynamic function type must themselves be dynamic. Neither are there side conditions in the typing rules, requiring for example that the variable bound in a dynamic **let** itself be of a dynamic type. Such extra conditions are an essential part of other two-level λ-calculi; they are needed to make partial evaluation possible. The key advantage of type specialisation is that these conditions can be dropped.

The standard semantics of the meta-language is just the usual semantics for the (call-by-name) λ-calculus: static and dynamic constructs are not distinguished, and both **poly** e and **spec** e have the same semantics as e. These are *annotations* to control specialisation, and as such do not affect

$$
\begin{array}{rcl}
e & ::= & \cdots \mid \eta\ e \mid \mathbf{mlet}\ x \Leftarrow e\ \mathbf{in}\ e \\
\tau & ::= & \cdots \mid M\ \tau
\end{array}
$$

$$
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \eta\ e : M\ \tau} \qquad \frac{\Gamma \vdash e_1 : M\ \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : M\ \tau_2}{\Gamma \vdash \mathbf{mlet}\ x \Leftarrow e_1\ \mathbf{in}\ e_2 : M\ \tau_2}
$$

Figure 3: The Computational Metalanguage

the meaning of the original program.

Type specialisation can be specified quite simply via *specialisation rules* that let us infer judgements of the form

$$\Gamma \vdash e : \tau \hookrightarrow e' : \tau'$$

where the context Γ tells us how to specialise variables, via a sequence of entries of the form $x : \sigma \hookrightarrow e : \sigma'$. A selection of specialisation rules appears in Figure 2; we have omitted the rules for static functions, which are a little more complicated to express. Compare the rules for static and dynamic **let** expressions: the *only* difference is that a static **let** is unfolded, while a dynamic one is not. Precisely the same static information about the bound variable x is available in both cases. Again, this is in contrast to partial evaluation, where a variable bound by a dynamic **let** must itself be dynamic.

The specialisation rules are more complicated to implement than the usual type inference rules, because they are not all syntax-directed. A description of the implementation is beyond the scope of this paper, but can be found elsewhere [17].

3 A Metalanguage with Effects

Languages with effects are invariably harder to treat formally than purely functional languages. One reason is that evaluation order suddenly becomes important, even for constructs in the 'pure' subset. Adding references and unrestricted assignment to our metalanguage would therefore change the semantics of every construct, and potentially invalidate all the specialisation rules presented above.

To avoid this, we have chosen to work with Moggi's computational metalanguage Λ_{ml} [25], which explicitly distinguishes between a value of type τ, and a computation (with effects) delivering a result of type τ. The latter is assigned a *computation type* $M\ \tau$. Trivial computations (with no effects) can be created by the *unit* operator η, and the results of computations can be inspected using $\mathbf{mlet}\ x \Leftarrow e_1\ \mathbf{in}\ e_2$, which combines the computations e_1 and e_2 in sequence, and binds x to the value that e_1 delivers in the evaluation of e_2. The extensions to the syntax and type-system of our metalanguage are shown in Figure 3.

If e_1 and e_2 are computations with effects, then it is a type error to write $e_1 + e_2$. Instead, we must write

$$\mathbf{mlet}\ x_1 \Leftarrow e_1\ \mathbf{in}\ \mathbf{mlet}\ x_2 \Leftarrow e_2\ \mathbf{in}\ \eta\ (x_1 + x_2)$$

making the evaluation order explicit. Similarly, to pass the result of e_1 to a function f, we must write $\mathbf{mlet}\ x \Leftarrow e_1\ \mathbf{in}\ f\ x$, making it explicit that e_1 is evaluated before the call. To write $f\ e_1$ instead would *not* compute e_1 first: it would pass the entire computation as a parameter to f, to be invoked later (or not) at f's pleasure. Think of a value of type $M\ \tau$ as a *suspended* computation, that can be invoked by **mlet**.

Many different kinds of effects can be represented in Moggi's framework, depending on how the types $M\ \tau$ and operators η and **mlet** are interpreted. To represent side effects on a store for example, we can interpet a computation as a function from the store beforehand to a result and the store afterwards:

$$
\begin{array}{rcl}
M\ \tau & = & \mathit{Store} \to (\tau, \mathit{Store}) \\
\eta\ e & = & \lambda s.(e, s) \\
\mathbf{mlet}\ x \Leftarrow e_1\ \mathbf{in}\ e_2 & = & \lambda s.\mathbf{let}\ (x, s') = e_1\ s\ \mathbf{in}\ e_2\ s'
\end{array}
$$

Note that the translation passes the store that e_1 produces to e_2, and that x is in scope in e_2. Operations to create and assign to references are interpreted as suitable functions with a computation type. An interpretation for M, η and **mlet** is called a *monad*, provided it satisfies the so-called *monad laws*:

$$
\begin{array}{rcl}
\mathbf{mlet}\ x \Leftarrow \eta\, e_1\ \mathbf{in}\ e_2 & = & \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \\
\mathbf{mlet}\ x \Leftarrow e\ \mathbf{in}\ \eta\ x & = & e
\end{array}
$$

$$
\begin{array}{c}
\mathbf{mlet}\ x \Leftarrow (\mathbf{mlet}\ y \Leftarrow e_1\ \mathbf{in}\ e_2)\ \mathbf{in}\ e_3 = \\
\mathbf{mlet}\ y \Leftarrow e_1\ \mathbf{in}\ \mathbf{mlet}\ x \Leftarrow e_2\ \mathbf{in}\ e_3
\end{array}
$$

where in the third law, y must not occur free in e_3.

Haskell provides imperative operations in exactly this way, but of course other imperative languages do not distinguish computations explicitly. However, it is well known that a λ-calculus with implicit effects can be simply translated (in several different ways) into Λ_{ml}, each translation making a different evaluation order explicit [15]. By extending the call-by-value translation we can map, say, ML or Algol programs into our metalanguage. Thus we claim that type specialisation is also applicable to such languages; they simply have to mapped into an intermediate form which makes evaluation order explicit first.

Moggi's language is a conservative extension of the λ-calculus; the meaning of the pure subset remains unchanged. Isomorphisms such as $\mathbf{void} \to \tau \simeq \tau$ remain true — if a function with this type has effects, then τ must be a computation type, and *both* sides represent a suspended computation. This is its big advantage for us: the type specialisation and void erasure rules for the pure metalanguage remain valid when effects are added. We need only define new rules for the new constructs.

The specialisation rules for η and **mlet** cannot be given yet: they depend on which kind of effects we are interested in. But we can note that these operations do *not* come in static and dynamic versions. We will indeed be interested in both static and dynamic effects, but since both kinds may be used in one and the same computation, we need only one computation type, and therefore only one version of the monad operators.

$$\begin{array}{rcl} e & ::= & \cdots \mid \underline{\mathbf{ref}}\ e \mid \underline{\mathbf{!}}\ e \mid e \mathrel{\underline{\mathbf{:=}}} e \\ \tau & ::= & \cdots \mid \underline{\mathbf{ref}}\ \tau \end{array}$$

$$\frac{\Gamma \models e : \tau}{\Gamma \models \underline{\mathbf{ref}}\ e : M\ \underline{\mathbf{ref}}\ \tau} \qquad \frac{\Gamma \models e : \underline{\mathbf{ref}}\ \tau}{\Gamma \models \underline{\mathbf{!}}\ e : M\ \tau} \qquad \frac{\Gamma \models e_1 : \underline{\mathbf{ref}}\ \tau \quad \Gamma \models e_2 : \tau}{\Gamma \models e_1 \mathrel{\underline{\mathbf{:=}}} e_2 : M\ \mathbf{void}}$$

Figure 4: Operations on Dynamic References

$$\tau' \quad ::= \quad \cdots \mid \mathbf{ref}\ \tau' \quad \text{residual types}$$

$$\frac{\Gamma \models e : \tau \hookrightarrow e' : \tau'}{\Gamma \models \eta\ e : M\ \tau \hookrightarrow \eta\ e' : M\ \tau'}$$

$$\frac{\Gamma \models e_1 : M\ \tau_1 \hookrightarrow e_1' : M\ \tau_1' \quad \Gamma, x : \tau_1 \hookrightarrow x' : \tau_1' \models e_2 : M\ \tau_2 \hookrightarrow e_2' : M\ \tau_2'}{\Gamma \models \mathbf{mlet}\ x \ \Leftarrow\ e_1\ \mathbf{in}\ e_2 : M\ \tau_2 \hookrightarrow \mathbf{mlet}\ x' \ \Leftarrow\ e_1'\ \mathbf{in}\ e_2' : M\ \tau_2'}\ x' \text{ fresh}$$

$$\frac{\Gamma \vdash e : \tau \hookrightarrow e' : \tau'}{\Gamma \vdash \underline{\mathbf{ref}}\ e : M\ (\underline{\mathbf{ref}}\ \tau) \hookrightarrow \mathbf{ref}\ e' : M\ (\mathbf{ref}\ \tau')}$$

$$\frac{\Gamma \vdash e : \underline{\mathbf{ref}}\ \tau \hookrightarrow e' : \mathbf{ref}\ \tau'}{\Gamma \vdash \underline{\mathbf{!}}\ e : M\ \tau \hookrightarrow \mathbf{!}\ e' : M\ \tau'}$$

$$\frac{\Gamma \vdash e_1 : \underline{\mathbf{ref}}\ \tau \hookrightarrow e_1' : \mathbf{ref}\ \tau' \quad \Gamma \vdash e_2 : \tau \hookrightarrow e_2' : \tau'}{\Gamma \vdash e_1 \mathrel{\underline{\mathbf{:=}}} e_2 : M\ \mathbf{void} \hookrightarrow e_1' \mathrel{\mathbf{:=}} e_2' : M\ \mathbf{void}}$$

Figure 5: Specialisation Rules for Dynamic State

4 Specialising Operations on State

We approach the specialisation of imperative operations in three steps. First, we consider the situation when all these operations are deferred until run-time. This is already enough to achieve optimal specialisation for a self-interpreter. Then in Sec. 4.3, we move on to consider static effects. Finally in section 4.4 we show how to combine the two. This requires us to factorise the monad M into a static state transformer and a second monad, so that the specialiser can manipulate the static state.

4.1 Specialising a Dynamic State

We shall extend the metalanguage with operations to create, inspect, and update dynamic references. The additional syntax and typing rules are given in Figure 4; $\underline{\mathbf{ref}}$ creates a reference, $\underline{\mathbf{!}}$ reads its contents, and $\underline{\mathbf{:=}}$ assigns to it. Of course, all three operations have a computation type.

Under the assumption that all monadic operations are dynamic the specialisation rules turn out to be rather straightforward (see Fig. 5). The monadic operations simply specialise their subexpressions and reconstruct themselves. In spite of this simplicity it is still possible to achieve interesting results, since dynamic references can contain static information in their residual type, which is propagated to the rest of the computation via unification. For example, specialising

$$\begin{array}{ll} \mathbf{mlet}\ r \Leftarrow \underline{\mathbf{ref}}\ 2 & : M\ \underline{\mathbf{int}} \\ \mathbf{in}\quad \mathbf{mlet}\ x \Leftarrow \underline{\mathbf{!}}\ r & \\ \qquad \mathbf{in}\ \ \eta\ (\mathbf{lift}\ (x+1)) & \end{array}$$

yields

$$\begin{array}{ll} \mathbf{mlet}\ r \Leftarrow \underline{\mathbf{ref}}\ \bullet & : M\ \mathbf{int}. \\ \mathbf{in}\quad \mathbf{mlet}\ x \Leftarrow \underline{\mathbf{!}}\ r & \\ \qquad \mathbf{in}\ \ \eta\ 3 & \end{array}$$

The static value 2 is not present at run-time, but is preserved in the residual type of r (which is $\mathbf{ref}\ 2$). When r is dereferenced we know statically that the result is 2, even though r itself is dynamic.

We can see from the specialised expression that the reference r still exists in the residual program, but it is useless because its contents are of **void** type. The final phase of type specialisation, void erasure, is extended to remove such references and operations on them. It will thus transform the above term to $\eta\ 3$. (We assume here that reading and writing are the only operations on references; if testing references for equality were also available then erasing **void** references would not be correct.)

Since the residual type of a dynamic reference includes the residual type of the value it contains, *all the values we assign to it must have the same static part.* In the example above, the only value we can assign to r is 2! When a dynamic reference contains a partially static value, then the static part must be the same at each assignment, but the dynamic part can vary. For example, a reference of type $\underline{\mathbf{ref}}\ \mathit{Univ}$ (where *Univ* was defined in Sec. 2) created by

$$\mathbf{mlet}\ r\ \Leftarrow\ \underline{\mathbf{ref}}\ (\mathit{Num}\ (\mathbf{lift}\ 2))\ \mathbf{in}\ \ldots$$

could later be assigned $\mathit{Num}\ (\mathbf{lift}\ 3)$ or $\mathit{Num}\ (\mathbf{lift}\ 4)$, because all these values have residual type $\mathit{Num}\ \mathbf{int}$. It could *not* be assigned a value of the form $\mathit{Fun}\ f$. The residual program would just contain a reference to an **int**, which would be assigned the (untagged) values 2, 3 and 4.

When we want a reference to contain many different *static* values, we are obliged to use a static reference. The situation is comparable to the difference between static and dynamic functions: a dynamic function can be specialised to only *one* static argument, whereas a static function (which is unfolded) can be applied to any number.

4.2 Optimal Specialisation

This specialiser is already capable of optimal specialisation. To show this, we give a self-interpreter for the meta-language in Figure 6. (The interpreter is recursive: see [17] for specialisation rules for recursion). Since the interpreted language is typed, we represent values by a *static* sum type (*Univ*). When an expression of type *Univ* is specialised, its residual type (for example *Num* **int** or *Fun* (*Num* **int** $\to$ *Num* **int**)) tells us what type of value it represents. All type tags are known at specialisation time, and so both they and the corresponding tag tests are simplified away by the specialiser. As a consequence, this interpreter can only be specialised to *well-typed* programs — supplying an ill-typed input will lead to a failure during residual type inference.

No static references are needed to write the self-interpreter, because the only use of references in the interpreter is to model references in the interpreted program. Such references can only be assigned values of one type (because the interpreted language is well-typed), and so during specialisation all the values assigned will have the same static part.

When we specialise this interpreter to a term, we obtain essentially the same term as the result, up to α-conversion. The only operations in the interpreter which give rise to residual code are those marked dynamic, the **lift** in the interpretation of constants, and the monadic operations **mlet** and η. By inspection, it is clear in most cases that *eval* specialises to a copy of its second argument, provided the same is true for the recursive calls. For example, the case for constants contains one **lift**, which generates a residual constant; the case for λ-expressions contains one dynamic λ, which generates a residual λ-expression, and so on. Only two cases break this pattern: those for *Rf* and *As* each generate an additional **mlet** and η. However, these can always be removed by a post-processor which applies the monad laws. In the case of *Rf*, the residual code is always of the form $\mathbf{mlet}\ r\ \Leftarrow\ \mathbf{ref}\ e\ \mathbf{in}\ \eta\ r$, which by the second monad law is equal to $\mathbf{ref}\ e$. In the case of *As*, the residual code is of the form $\mathbf{mlet}\ z\ \Leftarrow\ e_1 := e_2\ \mathbf{in}\ \eta\ \bullet$. Since z is of type **void** this is equivalent to $\mathbf{mlet}\ z\ \Leftarrow\ e_1 := e_2\ \mathbf{in}\ \eta\ z$, which is equal to $e_1 := e_2$ by the second monad law. The postprocessor performs both these simplifications. Specialisation of the interpreter is therefore optimal.

(In fact the post-processor may simplify additional redexes, so to be more accurate: when we specialise this interpreter to a term we obtain a *reduct* of the original term. We claim that this specialisation should still be considered optimal — it would be unreasonable to refuse to do so because our specialiser sometimes performs *more* simplifications than optimality requires!)

4.3 Specialising a Static State

Dynamic references are created and updated at run-time. Consequently we cannot know their contents during specialisation, and so we insist that all their contents have the *same* static part, which we do know at specialisation time. So for example, the expression

$$\mathbf{mlet}\ r\ \Leftarrow\ \underline{\mathbf{ref}}\ 2\ \mathbf{in}\ \mathbf{mlet}\ z\ \Leftarrow\ r\ \underline{:=}\ 3\ \mathbf{in}\ \underline{!}\ r$$

would cause a specialisation error, since r is used with both **ref** 2 and **ref** 3 as residual types. We could avoid the error by making the reference contents dynamic:

$$\mathbf{mlet}\ r\ \Leftarrow\ \underline{\mathbf{ref}}\ \mathbf{lift}\ 2\ \mathbf{in}\ \mathbf{mlet}\ z\ \Leftarrow\ r\ \underline{:=}\ \mathbf{lift}\ 3\ \mathbf{in}\ \underline{!}\ r$$

Now specialisation succeeds since r has residual type **ref int**, but on the other hand no useful simplification is performed.

In constrast, static references are created and updated during specialisation; the specialiser keeps track of their contents in a static state, and so they may freely be assigned values with *different* residual types at different times. If our example is rewritten using static references

$$\mathbf{mlet}\ r\ \Leftarrow\ \mathbf{ref}\ 2\ \mathbf{in}\ \mathbf{mlet}\ z\ \Leftarrow\ r := 3\ \mathbf{in}\ !\ r$$

then specialisation succeeds and produces

$$\eta\ \bullet : M\ 3$$

In this section we shall consider specialisation of a language with *only* static references. The new operations and their types are analogous to those for dynamic references in Figure 5, the only difference being that we omit the underlines.

When we specialised a dynamic state above, we were able to treat the monad and its operations as a black box: because all effects were deferred to run-time, the specialiser did not need to know anything about them. But now that we are treating static effects that is no longer the case. Let us therefore say explicitly that the computation type is $\mathit{State} \to (\tau, \mathit{State})$ — computations are represented as functions.

The static state itself is a tuple of the contents of the existing static references. We actually build up tuples out of pairs, so let us define a little syntactic sugar for values and types:

$$\begin{aligned} \langle x_1 x_2 \ldots x_n \rangle &= (x_1, (x_2, \ldots (x_n, \bullet))) \\ \langle \sigma_1 \sigma_2 \ldots \sigma_n \rangle &= (\sigma_1, (\sigma_2, \ldots (\sigma_n, \mathbf{void}))) \end{aligned}$$

We will also write $\lambda\langle x_1 \ldots x_n\rangle.e$; it should be obvious how to translate such pattern matching into combinations of projections.

In source programs the static state is implicit, but in residual programs we will represent it explicitly as a tuple, and pass it explicitly to and from residual computations. But now we encounter a problem. The residual type of the static state is a tuple type, whose components are the residual types of the contents of references. But a computation may modify those residual types! Thus, in the residual program, a computation may modify not only the *contents* of the state, but its *type*! Residual computations have types of the form

$$ST\ \sigma\ \sigma'\ \tau = \sigma \to (\tau, \sigma')$$

where σ is the residual type of the state beforehand, and σ' its type afterwards. This type is not a monad, which means that we cannot express the residual program in terms of the monad operators. However, ST has much in common with a monad — we can for example give sensible definitions of

$$
\begin{array}{l}
\textbf{data}\ E = \ \ Cn\ \textbf{int} \mid Vr\ \textbf{int} \mid Lm\ \textbf{int}\ E \mid Ap\ E\ E \mid Fx\ E \mid Un\ E \mid Ml\ \textbf{int}\ E\ E \mid Rf\ E \mid Df\ E \mid As\ E\ E \\
\textbf{data}\ U = \ \ Num\ \underline{\textbf{int}} \mid Fun\ (U \underline{\to} U) \mid Ref\ (\underline{\textbf{ref}}\ U) \mid Void \mid Mon\ (M\,U) \mid Wrong \\
\textbf{type}\ Env = \textbf{int} \to U \\
\\
\textbf{let}\ upd = \lambda env.\lambda i.\lambda v.\lambda j.\textbf{if}\ i = j\ \textbf{then}\ v\ \textbf{else}\ env\ j\ \textbf{in} \\
\textbf{letrec}\ \ eval = \lambda env.\lambda e. \\
\qquad\qquad \textbf{case}\ e\ \textbf{of} \\
\qquad\qquad\quad Cn\ n \qquad\quad : \ Num\ (\textbf{lift}\ n) \\
\qquad\qquad\quad Vr\ i \qquad\quad : \ env\ i \\
\qquad\qquad\quad Lm\ i\ e \qquad : \ Fun\ (\underline{\lambda} v.eval\ (upd\ env\ i\ v) e) \\
\qquad\qquad\quad Ap\ e_1\ e_2 \quad\ : \ \ \textbf{case}\ eval\ env\ e_1\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\quad Fun\ f \ \ : \ \ f\,\underline{@}(eval\ env\ e_2) \\
\qquad\qquad\quad Fx\ e \qquad\quad : \ \ \textbf{case}\ eval\ env\ e\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\quad Fun\ f \ \ : \ \ \underline{\textbf{fix}}\ f \\
\qquad\qquad\quad Un\ e \qquad\quad : \ Mon(\eta\ (eval\ env\ e)) \\
\qquad\qquad\quad Ml\ i\ e_1\ e_2 \ : \ \ \textbf{case}\ eval\ env\ e_1\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\quad Mon\ m_1 \ \ : \ \ \textbf{mlet}\ v \ \Leftarrow\ m_1\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{case}\ eval\ (upd\ env\ i\ v)\ e_2\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad Mon\ m_2 \ \ : \ \ m_2 \\
\qquad\qquad\quad Rf\ e \qquad\quad : \ Mon(\textbf{mlet}\ r \ \Leftarrow\ \underline{\textbf{ref}}\ (eval\ env\ e)\ \textbf{in}\ \eta\ (Ref\ r)) \\
\qquad\qquad\quad Df\ e \qquad\quad : \ \ \textbf{case}\ eval\ env\ e\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\quad Ref\ r \ \ : \ \ Mon(\underline{!}\ r) \\
\qquad\qquad\quad As\ e_1\ e_2 \quad\ : \ \ \textbf{case}\ eval\ env\ e_1\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\quad Ref\ r \ \ : \ \ Mon(\textbf{mlet}\ z \ \Leftarrow\ r\ \underline{:=}\ eval\ env\ e_2\ \textbf{in}\ \eta\ Void) \\
\textbf{in}\ eval\ (\lambda i.Wrong)
\end{array}
$$

Figure 6: Self-Interpreter for Optimal Specialisation

η and **mlet**, for which the monad laws hold — and so we call it a *quasimonad*[2].

The specialisation rules for static references are given in Figure 7. The rules for η and **mlet** just generate code to pass the state around explicitly. Notice in the rule for **mlet** that the residual types propagate static information about the store from e_1 to e_2 (via the shared variable σ_1).

Creating a static reference (the third rule) enlarges the static store by one element. The static reference may very well have dynamic contents, which become part of the residual static store. The only static information we need about the new reference itself is the location to which it refers, so we introduce a new form of residual type loc_i to represent the ith location. The dereferencing and assignment operations just generate functions that access or modify the residual store appropriately. Notice how the residual types record precisely the static contents of every reference before and after each operation.

The last three rules cannot be applied unless the shape of the static store is known; the last two cannot either be applied unless the address of the static location i is also known. During specialisation this information should be propagated from the context by unification. It is considered a specialisation error if it is not possible to determine the size of the static store during specialisation.

For example, suppose we specialise the term

$$
\begin{array}{l}
\textbf{mlet}\ x \Leftarrow \textbf{ref}\ (\textbf{lift}\ 2) \\
\textbf{in}\ \ \ \textbf{mlet}\ y \Leftarrow \textbf{ref}\ 3 \\
\qquad \textbf{in}\ \ \textbf{mlet}\ z\ \Leftarrow\ x := \textbf{lift}\ 4\ \textbf{in}\ \eta\ (x, y) \\
: M\ (\textbf{ref}\ \underline{\textbf{int}}, \textbf{ref}\ \textbf{int})
\end{array}
$$

in an initially empty static store. The result of applying the

[2] This is quite different from Steele's *pseudomonads* [31].

specialisation rules is

$$
\begin{array}{l}
\lambda s_0.\ \textbf{let}\ (x, s_1) = (\lambda\langle\rangle.(\bullet, \langle 2\rangle))\ s_0 \\
\qquad \textbf{in}\ \ (\lambda s_1.\ \textbf{let}\ (y, s_2) = (\lambda\langle v_1\rangle.(\bullet, \langle v_1 \bullet\rangle))\ s_1 \\
\qquad\qquad\qquad \textbf{in}\ \ (\lambda s_2.\ \textbf{let}\ (z, s_3) = (\lambda\langle v_1 v_2\rangle.(\bullet, \langle 4\ v_2\rangle))\ s_2 \\
\qquad\qquad\qquad\qquad\quad \textbf{in}\ \ (\lambda s_3.((x, y), s_3))\ s_3)\ s_2)\ s_1 \\
: ST\ \langle\rangle\ \langle\textbf{int}\ 3\rangle\ (\mathrm{loc}_1, \mathrm{loc}_2)
\end{array}
$$

Now s_0, x, y and z have void types and can so be removed by the void eraser, which also deletes void components from tuples. The result of void erasure in this case is

$$
\begin{array}{l}
\textbf{let}\ s_1 = 2 \\
\textbf{in}\ \ (\lambda s_1.\ \textbf{let}\ s_2 = (\lambda v_1.v_1)\ s_1 \\
\qquad\quad \textbf{in}\ \ (\lambda s_2.\ \textbf{let}\ s_3 = (\lambda v_1.4)\ s_2 \\
\qquad\qquad\qquad \textbf{in}\ \ (\lambda s_3.s_3)\ s_3)\ s_2)\ s_1
\end{array}
$$

We use a post-processor which contracts trivial β-redexes (in which the actual parameter is just a variable), since the specialisation rules introduce many of these. The final result after post-processing is

$$\textbf{let}\ s_1 = 2\ \textbf{in}\ \textbf{let}\ s_2 = s_1\ \textbf{in}\ \textbf{let}\ s_3 = 4\ \textbf{in}\ s_3$$

All static operations on the store have been removed, and the resulting program just passes around the dynamic values which it held.

The residual programs we generate with this approach are purely functional, and pass around any dynamic components of the static store explicitly. An alternative approach which we considered was to store such dynamic components in the dynamic store instead. Thus every static reference (with dynamic contents) would be associated with a reference in the residual program, holding the dynamic part of its contents. But remember that a static reference can hold values with different residual types at different times — for example, a reference of type **ref** *Univ* could hold both *Num n* and *Fun f* on different occasions. No single residual reference can hold both dynamic parts, n and f, because they

$$\tau' \quad ::= \quad \cdots \mid \mathrm{loc}_n \quad \text{residual types}$$

$$\frac{\Gamma \models e : \tau \hookrightarrow e' : \tau'}{\Gamma \models \eta\ e : M\ \tau \hookrightarrow \lambda s.(e', s) : ST\ \sigma\ \sigma\ \tau'}\ s \text{ fresh}$$

$$\frac{\Gamma \models e_1 : M\ \tau_1 \hookrightarrow e_1' : ST\ \sigma_0\ \sigma_1\ \tau_1' \quad \Gamma, x : \tau_1 \hookrightarrow x' : \tau_1' \models e_2 : M\ \tau_2 \hookrightarrow e_2' : ST\ \sigma_1\ \sigma_2\ \tau_2'}{\Gamma \models \mathbf{mlet}\ x \ \Leftarrow\ e_1\ \mathbf{in}\ e_2 : M\ \tau_2 \hookrightarrow \lambda s_0.\mathbf{let}\ (x', s_1) = e_1'\ s_0\ \mathbf{in}\ e_2'\ s_1 : ST\ \sigma_0\ \sigma_2\ \tau_2'}\ x', s_0, s_1 \text{ fresh}$$

$$\frac{\Gamma \models e : \tau \hookrightarrow e' : \tau'}{\Gamma \models \mathbf{ref}\ e : M\ \tau \hookrightarrow \lambda\langle s_1 \ldots s_n\rangle.(\bullet, \langle s_1 \ldots s_n e'\rangle) : ST\ \langle \sigma_1 \ldots \sigma_n\rangle\ \langle \sigma_1 \ldots \sigma_n \tau'\rangle\ \mathrm{loc}_{n+1}}\ s_i \text{ fresh}$$

$$\frac{\Gamma \models e : \mathbf{ref}\ \tau \hookrightarrow e' : \mathrm{loc}_i}{\Gamma \models \mathbf{!}\ e : M\ \tau \hookrightarrow \lambda\langle s_1 \ldots s_n\rangle.(s_i, \langle s_1 \ldots s_n\rangle) : ST\ \langle \sigma_1 \ldots \sigma_n\rangle\ \langle \sigma_1 \ldots \sigma_n\rangle\ \sigma_i}\ s_i \text{ fresh}$$

$$\frac{\Gamma \models e_1 : \mathbf{ref}\ \tau \hookrightarrow e_1' : \mathrm{loc}_i \quad \Gamma \models e_2 : \tau \hookrightarrow e_2' : \tau'}{\begin{array}{l}\Gamma \models \ \ e_1 \mathrel{\mathbf{:=}} e_2 : M\ \mathbf{void}\ \hookrightarrow \\ \quad \lambda\langle s_1 \ldots s_n\rangle.(\bullet, \langle s_1 \ldots s_{i-1} e_2' s_{i+1} \ldots s_n\rangle) : ST\ \langle \sigma_1 \ldots \sigma_n\rangle\ \langle \sigma_1 \ldots \sigma_{i-1} \tau' \sigma_{i+1} \ldots \sigma_n\rangle\ \mathbf{void}\end{array}}\ s_i \text{ fresh}$$

Figure 7: Specialisation Rules for Static State

have different types. We could perhaps associate each static reference with a tuple of references in the residual program, one for each type of value it is assigned. At any dereference operation, we would of course know statically which of the associated residual references held the current dynamic part of the contents. But reference creation would be awkward, because all but one of the corresponding residual references would be uninitialised. The problems are probably soluble, but we feel that the approach we presented above is considerably simpler.

4.4 Specialising a Mixed State

Now that we know how to specialise a static and a dynamic state separately, it is relatively straightforward to specialise them together. The trick is to separate the monad into two parts: a static part which we manipulate explicitly in the specialiser, and a dynamic part which we treat as a black box. We therefore interpret the monad type $M\ \tau$ as $State \rightarrow M'\ (\tau, State)$, where M' is the monad type used in residual programs. As in the previous section, residual computations may change the type of the static state; they will have types of the form

$$ST\ \sigma\ \sigma'\ \tau = \sigma \rightarrow M'\ (\tau, \sigma')$$

The specification rules have to be modified to produce terms of this type. We modify the rules for dynamic reference operations to pass the static store on unchanged, and we modify the rules for static reference operations to construct trivial M' computations using η. The modified rules are given in Figure 8.

4.5 Specialisation of a Lazy Interpreter

We demonstrate the capabilities of the specialiser by specialising a lazy interpreter to a term. Laziness is implemented by using updatable closures. At application an abstraction is passed a reference to a closure, which when evaluated reduces the argument to weak-head normal form, and performs an update on the reference cell. Provided only a bounded number of closures are created, the reference operations can be performed statically. Figure 9 shows the interpreter.

Specialising the interpreter to the term $(\lambda x.x + x)(1 + 2)$ yields the residual type $ST\ \langle\rangle\ \langle Vl\ (Num\ \mathbf{int})\rangle\ (Num\ \mathbf{int})$. Here the output store contains the $Vl\ (Num\ \mathbf{int})$. The constructor Vl tells us that the corresponding closure was evaluated at specialisation time. The constructor Num is also manipulated at specialisation time, which means that we can eliminate tagging and untagging from residual programs. What remains to be stored at run time is the integer. The residual term after additional simplication (see section 6) is

$$\begin{array}{l}\mathbf{let}\ f = \lambda s.\mathbf{mlet}\ a\ \Leftarrow\ s\ s\ \mathbf{in}\ \eta(a + a, a) \\ \mathbf{in}\quad \mathbf{let}\ b = \lambda s.\eta(1 + 2) \\ \qquad \mathbf{in}\ \ f\ \ b\end{array}$$

The resulting program, although a bit awkward, has the expected structure. f corresponds to the abstraction, and b to the argument $1+2$, for which we have to build a closure. The closure b takes a store s, and returns the result of the addition. f takes this closure—dereference has taken place at specialisation time – and applies the closure to itself—the closure is the sole element in the store. The result of this application is a computation taken apart by the monadic let. We then perform the addition $a + a$, and tuple it with a, the result of updating the store. The application $s\ s$ of the store to itself may seem strange, but it has a perfectly admissible recursive type $\mu\alpha.\alpha \rightarrow M\ \mathbf{int}$.

5 Delimiting Effects

The type specialiser is able, among other things, to derive static information about the result of a dynamic conditional. For example,

$$\begin{array}{l}\underline{\mathbf{if}}\ b\ \underline{\mathbf{then}}\ Num\ (\mathbf{lift}\ 3)\ \underline{\mathbf{else}}\ Num\ (\mathbf{lift}\ 4) : Univ\ \hookrightarrow \\ \quad \mathbf{if}\ b'\ \mathbf{then}\ 3\ \mathbf{else}\ 4 : Num\ \mathbf{int}\end{array}$$

which tells us statically that the result is tagged Num. Of course, for this to be possible and for the residual program to be well typed, the two branches must have the *same* residual type. When the arms of the conditional are computations, this implies that they must leave the static store in the same state.

$$\frac{\Gamma \mathrel{|\!\!-} e : \tau \hookrightarrow e' : \tau'}{\Gamma \mathrel{|\!\!-} \eta\ e : M\ \tau \hookrightarrow \lambda s.\eta(e', s) : ST\ \sigma\ \sigma\ \tau'}\ s \text{ fresh}$$

$$\frac{\Gamma \mathrel{|\!\!-} e_1 : M\ \tau_1 \hookrightarrow e_1' : ST\ \sigma_0\ \sigma_1\ \tau_1' \quad \Gamma, x : \tau_1 \hookrightarrow x' : \tau_1' \mathrel{|\!\!-} e_2 : M\ \tau_2 \hookrightarrow e_2' : ST\ \sigma_1\ \sigma_2\ \tau_2'}{\Gamma \mathrel{|\!\!-} \mathbf{mlet}\ x\ \Leftarrow\ e_1\ \mathbf{in}\ e_2 : M\ \tau_2 \hookrightarrow \lambda s_0.\mathbf{mlet}\ (x', s_1)\ \Leftarrow\ e_1'\ s_0\ \mathbf{in}\ e_2'\ s_1 : ST\ \sigma_0\ \sigma_2\ \tau_2'}\ x', s_0, s_1 \text{ fresh}$$

$$\frac{\Gamma \mathrel{|\!\!-} e : \tau \hookrightarrow e' : \tau'}{\Gamma \mathrel{|\!\!-} \mathbf{ref}\ e : M\ \tau \hookrightarrow \lambda\langle s_1 \ldots s_n\rangle.\eta(\bullet, \langle s_1 \ldots s_n e'\rangle) : ST\ \langle \sigma_1 \ldots \sigma_n\rangle\ \langle \sigma_1 \ldots \sigma_n \tau'\rangle\ \mathrm{loc}_{n+1}}\ s_i \text{ fresh}$$

$$\frac{\Gamma \mathrel{|\!\!-} e : \mathbf{ref}\ \tau \hookrightarrow e' : \mathrm{loc}_i}{\Gamma \mathrel{|\!\!-} \mathbf{!}\ e : M\ \tau \hookrightarrow \lambda\langle s_1 \ldots s_n\rangle.\eta(s_i, \langle s_1 \ldots s_n\rangle) : ST\ \langle \sigma_1 \ldots \sigma_n\rangle\ \langle \sigma_1 \ldots \sigma_n\rangle\ \sigma_i}\ s_i \text{ fresh}$$

$$\frac{\Gamma \mathrel{|\!\!-} e_1 : \mathbf{ref}\ \tau \hookrightarrow e_1' : \mathrm{loc}_i \quad \Gamma \mathrel{|\!\!-} e_2 : \tau \hookrightarrow e_2' : \tau'}{\begin{array}{l}\Gamma \mathrel{|\!\!-}\ \ e_1\ \mathbf{:=}\ e_2 : M\ \mathbf{void}\ \hookrightarrow \\ \quad \lambda\langle s_1 \ldots s_n\rangle.\eta(\bullet, \langle s_1 \ldots s_{i-1} e_2' s_{i+1} \ldots s_n\rangle) : ST\ \langle \sigma_1 \ldots \sigma_n\rangle\ \langle \sigma_1 \ldots \sigma_{i-1} \tau' \sigma_{i+1} \ldots \sigma_n\rangle\ \mathbf{void}\end{array}}\ s_i \text{ fresh}$$

$$\frac{\Gamma \mathrel{|\!\!-} e : \tau \hookrightarrow e' : \tau'}{\Gamma \vdash \underline{\mathbf{ref}}\ e : M\ (\underline{\mathbf{ref}}\ \tau) \hookrightarrow \lambda s.\mathbf{mlet}\ r\ \Leftarrow\ \mathbf{ref}\ e'\ \mathbf{in}\ \eta(r, s) : ST\ \sigma\ \sigma\ (\mathbf{ref}\ \tau')}\ s, r \text{ fresh}$$

$$\frac{\Gamma \vdash e : \underline{\mathbf{ref}}\ \tau \hookrightarrow e' : \mathbf{ref}\ \tau'}{\Gamma \vdash \underline{\mathbf{!}}\ e : M\ \tau \hookrightarrow \lambda s.\mathbf{mlet}\ v\ \Leftarrow\ \mathbf{!}\ e'\ \mathbf{in}\ \eta(v, s) : ST\ \sigma\ \sigma\ \tau'}\ s, v \text{ fresh}$$

$$\frac{\Gamma \vdash e_1 : \underline{\mathbf{ref}}\ \tau \hookrightarrow e_1' : \mathbf{ref}\ \tau' \quad \Gamma \vdash e_2 : \tau \hookrightarrow e_2' : \tau'}{\Gamma \vdash e_1\ \underline{\mathbf{:=}}\ e_2 : M\ \mathbf{void} \hookrightarrow \lambda s.\mathbf{mlet}\ z\ \Leftarrow\ e_1'\ \mathbf{:=}\ e_2'\ \mathbf{in}\ \eta(z, s) : ST\ \sigma\ \sigma\ \mathbf{void}}\ s, z \text{ fresh}$$

Figure 8: Specialisation Rules for Mixed State

It is very awkward to meet this restriction. It means that if any references are created in one branch, then exactly corresponding references must be created in the other, and assigned contents with the same static part! Most partial evaluators for imperative languages handle the problem by forbidding modifications to the static store in the arms of a dynamic conditional, but this makes the static store much less useful. Our solution to the problem is to support a *stack discipline* for the static store. We introduce a 'store prompt' $\#\ e$, which causes the specialiser to deallocate static references created by e after its specialisation is complete (see Fig. 10). If we use store prompts in the arms of a dynamic conditional, then each arm can refer to the previous static store, and create and use its own local references, without causing a residual type mismatch. Of course store prompts should only be used when local references do not escape from the enclosed expression.

As we have explained, operations on the static store can only be specialised once the size of the store is known. The specialisation rules propagate the static store from operation to operation, so if the initial size of the store is known, then its size will be derivable at all subsequent operations. Of course we intend the initial store to be empty, but so far we have glossed over how we specify this.

In fact, there can be other reasons for wanting to specify the size of the store. Consider the function

$$\underline{\lambda} t.\mathbf{mlet}\ r\ \Leftarrow\ \mathbf{ref}\ 1\ \mathbf{in}\ \mathbf{mlet}\ v\ \Leftarrow\ \mathbf{!}\ r\ \mathbf{in}\ \eta\ (t \underline{+} \mathbf{lift} v).$$

which uses the static store internally to compute $t+1$. This function's type is $\underline{\mathbf{int}{\rightarrow}}M\ \underline{\mathbf{int}}$, and it cannot be specialised without knowing the size of the static store at the point where it is called. That seems unfortunate since it doesn't refer to any non-local references. Moreover, if the function were called from two points with different static stores, then we would need to make it polyvariant and construct two different specialisations — even though it doesn't access those different stores. This would be the case even if we inserted a store prompt into the function's definition.

The solution is to run the computation in the function's body in a new, empty static store. We define two new operators to do so, which differ only in their treatment of the *dynamic* store (see Fig. 11). $\underline{\mathbf{runM}}\ e$ turns a computation with effects into one without, from a computational perspective. It runs e in a completely new (static and dynamic) store, so that e can neither access nor modify the enclosing store — it is a state thread. The residual type does not involve the monad, and so the residual code must itself include the effect delimiter **runM** . **runM** e runs e of type $M\tau$ in a new and empty static store, but threads the dynamic store through, so that e *can* access and modify non-local dynamic references. The residual type is still $M\tau'$ because of the dynamic store. We should ensure that references neither escape from $\underline{\mathbf{runM}}\ e$ nor can they be imported from the outside (and analogously for **runM** e and static references). We gloss over this problem here because we can apply Launchbury and Peyton Jones's solution to a related problem [20, 21].

The specialisation rule for $\underline{\mathbf{runM}}\ e$ (see Fig. 12) applies the residual quasimonad of e to an empty initial static state, runs the resulting dynamic computation, and discards the final static state. The result is a value with residual type τ'. The rule for **runM** e also applies the residual quasimonad to an empty static state, and the final state σ' is again discarded. In contrast with $\underline{\mathbf{runM}}$, the resulting expression might still induce effects at run-time. For this reason we thread the resulting computation into the dynamic computation thread.

Now we can specialise our initial example in two ways:

$$\underline{\lambda} t.\mathbf{runM} \left(\begin{array}{ll} \mathbf{mlet}\ r \Leftarrow \mathbf{ref}\ 1 & \\ \mathbf{in} & \mathbf{mlet}\ v \Leftarrow \mathbf{!}\ r \\ & \mathbf{in}\ \ \eta\ (t \underline{+} \mathbf{lift} v) \end{array} \right) : \underline{\mathbf{int}{\rightarrow}}M\ \underline{\mathbf{int}}$$

$$\hookrightarrow \lambda t.\mathbf{runM}\ (\eta\ (t+1)) : \mathbf{int} \rightarrow M\ \mathbf{int}$$

$$
\begin{array}{l}
\textbf{data}\ E = \ \ Cn\ \textbf{int} \mid Vr\ \textbf{int} \mid Lm\ \textbf{int}\ E \mid Ap\ E\ E \mid Pl\ E\ E \\
\textbf{data}\ U = \ \ Num\ \underline{\textbf{int}} \mid Fun\ (U \underline{\rightarrow} U) \mid Ref\ (\underline{\textbf{ref}}\ U) \mid Wrong \\
\textbf{data}\ V = \ \ Bl\ \textbf{int} \mid Cl(M\ \textbf{void}) \mid Vl\ U \\
\textbf{type}\ Env = \textbf{int} \rightarrow U \\
\\
\textbf{let}\ upd = \lambda env.\lambda i.\lambda v.\lambda j.\textbf{if}\ i = j\ \textbf{then}\ v\ \textbf{else}\ env\ j\ \textbf{in} \\
\textbf{letrec}\ \ eval = \lambda env.\lambda e. \\
\qquad\qquad \textbf{case}\ e\ \textbf{of} \\
\qquad\qquad\quad Cn\ n \qquad\ \ : \ \eta\ (Num\ (\textbf{lift}\ n)) \\
\qquad\qquad\quad Vr\ i \qquad\ \ \ : \quad \textbf{case}\ env\ i\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\quad Ref\ p \ \ : \ \ \textbf{mlet}\ v \ \Leftarrow\ \textbf{!}\ p\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{case}\ v\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Vl\ v \quad : \ \ \eta\ v \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Cl\ m \ \ : \ \ \textbf{mlet}\ a \ \Leftarrow\ m\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \textbf{mlet}\ b \ \Leftarrow\ \textbf{!}\ p\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \ \textbf{case}\ b\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Vl\ v \ \ : \ \ \eta\ v \\
\qquad\qquad\quad Lm\ i\ e \qquad : \ \eta(Fun\ (\underline{\lambda} v.eval\ (upd\ env\ i\ v) e)) \\
\qquad\qquad\quad Ap\ e_1\ e_2 \ \ : \ \textbf{mlet}\ f \ \Leftarrow\ eval\ env\ e_1\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\ \ \textbf{mlet}\ a \ \Leftarrow\ \textbf{ref}\ Bl\ 0\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\ \ \textbf{mlet}\ b \ \Leftarrow\ a := Cl\ (\textbf{mlet}\ c \ \Leftarrow\ eval\ env\ e_2\ \textbf{in}\ a := Vl\ c)\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{case}\ f\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\quad\quad Fun\ f \ \ : \ \ f\ \underline{@}(Ref\ a) \\
\qquad\qquad\quad Pl\ e_1\ e_2 \ \ \ : \ \textbf{mlet}\ a \ \Leftarrow\ eval\ env\ e_1\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\ \ \textbf{mlet}\ b \ \Leftarrow\ eval\ env\ e_2\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{case}\ a\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\quad\quad Num\ a \ \ : \quad \textbf{case}\ b\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Num\ b \ \ : \ \ \eta\ (Num\ a + b) \\
\textbf{in}\ eval\ (\lambda i.Wrong)
\end{array}
$$

Figure 9: Lazy Interpreter

$$
\frac{\Gamma \mid\!\!- e : M\ \tau}{\Gamma \mid\!\!- \#\ e : M\ \tau}
\qquad
\frac{\Gamma \vdash e : M\ \tau \hookrightarrow e' : ST\ \langle \sigma_1 \ldots \sigma_n \rangle\ \langle \sigma'_1 \ldots \sigma'_{n+k} \rangle\ \tau'}{\Gamma \vdash \#\ e : M\ \tau \hookrightarrow e' : ST\ \langle \sigma_1 \ldots \sigma_n \rangle\ \langle \sigma'_1 \ldots \sigma'_n \rangle\ \tau'}
$$

Figure 10: Typing and Specialisation Rules for Store Prompts

or

$$
\underline{\lambda} t.\underline{\textbf{runM}} \left(\begin{array}{l} \textbf{mlet}\ r \Leftarrow \textbf{ref}\ 1 \\ \textbf{in} \quad \textbf{mlet}\ v \Leftarrow \textbf{!}\ r \\ \qquad \textbf{in} \ \ \eta\ (t \underline{+} \textbf{lift} v) \end{array} \right) : \underline{\textbf{int} \rightarrow \textbf{int}}
$$

$$
\hookrightarrow \lambda t.t + 1 : \textbf{int} \rightarrow \textbf{int}
$$

The only other specialiser to combine first-class functions with static references is due to Dussart and Thiemann [13]. It is a more conventional partial evaluator, in which dynamic λ-expressions cannot have static arguments. Consequently it always specialises the bodies of dynamic λ-expressions in an empty static store. Our **runM** essentially simulates this behaviour.

Our effect delimiters **runM** and $\underline{\textbf{runM}}$ are rather blunt instruments. Ideally we would like to be able to specialise a computation knowing only the contents of static references that it actually refers to. Likewise we would like to be able to invoke such a specialisation in *different* static stores, provided the references actually used have the same contents. To do so requires polymorphism in residual programs — we want to invoke the same residual function on stores with different residual types. The generation of polymorphic programs by type specialisation is so far an open problem, which must be solved before a better treatment of static stores is possible.

6 Postprocessing

As we remarked above, Λ^{σ}_{ml} is a pure language in the sense that isomorphisms like $\textbf{void} \rightarrow \tau \simeq \tau$ are still valid. Therefore the all techniques for void erasure in the lambda calculus [17] are still valid.

6.1 Void Erasure

The void eraser follows one simple principle: Replace every expression whose type denotes a singleton set by $\bullet$. Apart from the standard situations involving products, sums, and function types described by Hughes [17], we now have two additional type constructors to consider: **ref** τ and $M\ \tau$. Keeping references of type **ref void** makes no sense unless there is an equality operation on references. They can be erased in the absence of such an operation. However, the type M **void** denotes a computation returning **void**, but the computation may very well affect the (dynamic) state. It cannot be erased.

$$\frac{\Gamma \mid\!- e : M\ \tau}{\Gamma \mid\!- \underline{\mathbf{runM}}\ e : \tau} \qquad \frac{\Gamma \mid\!- e : M\ \tau}{\Gamma \mid\!- \mathbf{runM}\ e : M\ \tau}$$

Figure 11: Type Rules for Store Delimiters

$$\frac{\Gamma \mid\!- e : M\ \tau \hookrightarrow e' : ST\ \langle\rangle\ \sigma'\ \tau'}{\Gamma \mid\!- \underline{\mathbf{runM}}\ e : \tau \hookrightarrow \pi_1(\mathbf{runM}\ (e'\ \langle\rangle)) : \tau'}$$

$$\frac{\Gamma \mid\!- e : M\ \tau \hookrightarrow e' : ST\ \langle\rangle\ \sigma'\ \tau'}{\Gamma \mid\!- \mathbf{runM}\ e : M\ \tau \hookrightarrow \lambda s.\mathbf{mlet}\ (x', s') \ \Leftarrow\ e'\ \langle\rangle\ \mathbf{in}\ \eta\ (x', s) : ST\ \sigma\ \sigma\ \tau'}$$

Figure 12: Specialisation Rules for Store Delimiters

Since we regard **ref void** as useless we can replace all operations processing them by appropriate monadic units:

$$\begin{array}{lcl} \mathbf{ref}\ e : M\ (\mathbf{ref\ void}) & \hookrightarrow & \eta\ \bullet : M\ \mathbf{void} \\ \mathbf{!}\ e : M\ \mathbf{void} & \hookrightarrow & \eta\ \bullet : M\ \mathbf{void} \\ e_1 : \mathbf{ref\ void} \Rightarrow e_1 := e_2 : M\ \mathbf{void} & \hookrightarrow & \eta\ \bullet : M\ \mathbf{void} \end{array}$$

This process removes operations on the dynamic state. The residual operations on the static state are expressed by explicit store passing. The simplification rules for the standard types suffice to simplify them.

6.2 Monadic Simplifications

The residual programs are written in monadic style and contain a lot of redundant monadic let and unit expressions. This is partly due to the actions of the preceding void erasure. These expressions present ideal targets for applying the three monad laws. Moggi [25] has proved that Λ_{ml} is sound and complete with respect to all monadic models, specifically for our model that uses state transformers. Therefore, the application of the monad laws is **very safe** in the sense that it does not duplicate, discard, or reorder computations.

If the type of x is **void** then after application of the rule

$$\mathbf{mlet}\ x\ \Leftarrow\ \eta\ e'\ \mathbf{in}\ e'' \hookrightarrow \mathbf{let}\ x = e'\ \mathbf{in}\ e''$$

we can drop the **let** and e' entirely if the type of x is **void**. In this case **let** $x = e'$ **in** $e'' \hookrightarrow e''$. The post-processor does not unfold **let**s in general, but we do use the simplication **let** $x = x$ **in** $e \hookrightarrow e$ because terms of this form appear quite often after applying the preceding transformations.

Finally, simplification discards trivial computations wrapped in **runM** :

$$\mathbf{runM}\ (\eta\ e) \hookrightarrow e.$$

6.3 Further Simplifications

Due to the explicit store-passing style, the residual program contains many trivial β-redexes like $(\lambda x.e)\ x$. We convert them to let-expressions and apply the same simplifications as outlined above. No other (serious) β reductions are performed.

The explicit representation of the store as tuples introduces a lot of extraneous tupling and tuple operations. These are eliminated via an arity raising phase [29].

Finally, explicit store-passing introduces many η-redexes on top of computations that do not affect the static store. Therefore, the void eraser also performs η-reductions.

7 Assessment

Type specialisation enhances partial evaluation with information propagation by unification. This enables side-ways information exchange in the source program's execution tree whereas partial evaluation is restricted to paths in the execution tree. A consequence is that residual programs cannot be constructed in a bottom-up manner, but rather program fragments are constructed in a demand-driven order prescribed by the flow of information.

Much of the power of type specialisaton stems from the refined type language used. The type language is able to express single element types that play the rôle of symbolic values and partially static values of standard partial evaluation.

In contrast to partial evaluation, the two-level type discipline of type specialisation does not impose well-formedness conditions like "if an abstraction is dynamic then its argument and result must also be dynamic." Another instance is the fact that dynamic references may hold static values. This liberality is unique to type specialisation. It is the key to the optimal specialisation results for typed functional languages reported here and in Hughes initial work [17], because it allows information (such as static tags) to be communicated out of dynamic functions.

However, the well-formedness conditions are at the heart of traditional binding-time analysis. A binding-time analyser constructs the least (most static) assignment of well-formed types to terms; dynamic inputs force other computations to be dynamic also via the well-formedness conditions. The most static type assignment is always 'best' — it leads to the strongest specialisation. The type specialiser in contrast can check that a two-level program is well-typed, but cannot infer a 'best' assignment of two-level types to a one-level program. In the absence of well-formedness constraints many more operations *can* be made static, but this does not mean that they *should* be. For example, in the interpreters in this paper we chose to make *Univ* a static sum type, so that type tags would be removed during specialisation. In consequence these interpreters can only be specialised to *well-typed* programs — they interpret a typed language.

Had we chosen to make *Univ* a dynamic sum instead, then type tags would have remained in residual programs, but we would be able to specialise the interpreters to arbitrary programs — such interpreters accept an untyped language. Neither alternative is obviously better than the other, and we do not believe the choice can be made by an automated analysis. The assignment of binding-times in an interpreter is essentially a way to specify the static semantics of the interpreted language, and this is a creative activity.

Furthermore, type specialisation also performs program analysis of the residual program while it constructs the residual program. An example for this is shown in Hughes work [17] where he constructs a firstifying two-level interpreter for a type lambda calculus. Here the type specializer performs closure analysis and the construction of recursive residual types on the fly.

8 Related Work

8.1 Type and Constructor Specialisation

Hughes [17] introduces type specialisation for a simply-typed lambda calculus with products and sums. The current work is an conservative extension to Moggi's computational metalanguage [25]. The type specialiser embodies constructor specialisation [24, 12] and extends it to higher-order languages. Hagiya and Iino [14] consider a weaker variant of constructor specialisation that they call "data type specialisation" for a Lisp-like language. They have set up an extension of the standard partial evaluation framework consisting of an extended binding-time analysis and a specialiser which is expressed as a non-standard interpreter and constructs the residual program in a compositional manner. Bechet [3] gives an account of a very similiar technique geared towards typed languages. The other constructor specialisers and the type specialiser construct the residual program out of order.

8.2 Partial Evaluation

Partial evaluators for traditional imperative languages like FORTRAN, C, Pascal, or Modula-2 [1, 10, 2, 8, 23] all perform side effects at specialization time. However, none of them incurs the problems inherent in higher-order control flow and only some of them face the problems of dealing with pointers which are similar to the problems with first-class references. In the present work, type specialisation deals satisfactorily with a language with higher-order control flow and first-class references.

Partial evaluators for higher-order functional languages are quite common [5, 7, 9, 4, 22, 32]. Only some of them attempt to handle side effects [7, 4, 22] but all of them follow the approach of Bondorf and Danvy [6] in deferring all side effects like operations on references, assignments to global variables, and I/O to run time.

Recently, there have been approaches to perform partial evaluation of imperative programs with static operations on references. Moura, Consel, and Lawall [26] show how to use Schism [9]—a partial evaluator for pure Scheme with a poyvariant binding-time analysis—to specialise programs in the C language by transforming them to a sophisticated variant of store-passing style. This approach avoids the construction of new program analyses and new partial evaluation techniques for imperative programs.

Dussart and Thiemann [13] have constructed an offline partial evaluator for a simply-typed lambda calculus with first-class references. Their system is traditional in that its monovariant binding-time analysis ensures that there are no errors at specialization time. The binding-time analysis is based on an effect system. The system is automatic, in contrast to the type specialiser which only works on manually annotated programs.

9 Conclusion

Type specialisation adapts seamlessly to the specialisation of imperative languages. Unification again gives us the necessary power to achieve optimal specialisation for a version of Moggi's computational metalanguage with first-class references.

Our use of the computational metalanguage Λ^{σ}_{ml} turned out to be a big win. First, we only had to add rules for the new constructs of the language to the type specializer. No existing rule was changed. Contrast this with the specializer of Dussart and Thiemann [13] where *every* specialization rule had to be changed in order to thread the state through the computation. Second, our specialiser is completely independent of the evaluation order. Only the sequence of the monadic operations is fixed; the remaining parts of a program can be specialized without committing to a specific evaluation order. Third, we conjecture that the monadic approach allows further modular extensions to cater for exceptions, non-determinism, etc.

Acknowledgement

We thank the anonymous reviewers and Mads Tofte for their valuable comments.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994.

[2] R. Baier, R. Glück, and R. Zöchling. Partial evaluation of numerical programs in Fortran. In P. Sestoft and H. Søndergaard, editors, *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM '94*, pages 119–132, Orlando, Fla., June 1994. ACM.

[3] D. Bechet. Removing value encoding using alternative values in partial evaluation of strongly-typed languages. In H. R. Nielson, editor, *Proc. 6th European Symposium on Programming*, pages 77–91, Linköping, Sweden, Apr. 1994. Springer-Verlag. LNCS 1058.

[4] L. Birkedal and M. Welinder. Hand-writing program generator generators. In M. V. Hermenegildo and J. Penjam, editors, *Programming Languages, Implementations, Logics, and Programs (PLILP '94)*, volume 844 of *Lecture Notes in Computer Science*, pages 198–214, Madrid, Spain, Sept. 1994. Springer-Verlag.

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

[6] A. Bondorf and O. Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Programming*, 16(2):151–195, 1991.

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

[8] M. A. Bulyonkov and D. V. Kochetov. Practical aspects of specialization of Algol-like programs. In Danvy et al. [11], pages 17–32.

[9] C. Consel. A tour of Schism. In Schmidt [30], pages 134–154.

[10] C. Consel and F. 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., Jan. 1996. ACM Press.

[11] O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, Dagstuhl, Germany, Feb. 1996. Springer Verlag, Heidelberg.

[12] D. Dussart, E. Bevers, and K. De Vlaminck. Polyvariant constructor specialization. In W. Scherlis, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '95*, pages 54–63, La Jolla, CA, June 1995. ACM Press.

[13] D. Dussart and P. Thiemann. Partial evaluation for higher-order languages with state. Berichte des Wilhelm-Schickard-Instituts WSI-96-??, Universität Tübingen, Nov. 1996.

[14] M. Hagiya and K. Iino. Binding time analysis for data type specialization. In M. Takeichi, editor, *Fuji Workshop on Functional and Logic Programming*, pages 254–269, Fuji Susono, Japan, July 1995. World Scientific Press, Singapore.

[15] J. Hatcliff and O. Danvy. A generic account of continuation-passing styles. In POPL1994 [28], pages 458–471.

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

[17] J. Hughes. Type specialisation for the λ-calculus; or, a new paradigm for partial evaluation based on type inference. In Danvy et al. [11], pages 183–215.

[18] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[19] J. Launchbury. A strongly-typed self-applicable partial evaluator. In Hughes [16], pages 145–164. LNCS 523.

[20] J. Launchbury and S. 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.

[21] J. Launchbury and S. L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, Dec. 1995.

[22] K. Malmkjær, O. Danvy, and N. 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.

[23] U. Meyer. Techniques for partial evaluation of imperative languages. In P. Hudak and N. D. Jones, editors, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '91*, pages 94–105, New Haven, CT, June 1991. ACM. SIGPLAN Notices 26(9).

[24] T. Æ. Mogensen. Constructor specialization. In Schmidt [30], pages 22–32.

[25] E. 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.

[26] B. Moura, C. Consel, and J. Lawall. Bridging the gap between functional and imperative languages. Publication interne 1027, Irisa, Rennes, France, July 1996.

[27] F. Nielson and H. R. Nielson. *Two-Level Functional Languages*. Cambridge University Press, 1992.

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

[29] S. A. Romanenko. Arity raiser and its use in program specialization. In N. D. Jones, editor, *Proc. 3rd European Symposium on Programming 1990*, pages 341–360, Copenhagen, Denmark, 1990. Springer-Verlag. LNCS 432.

[30] D. Schmidt, editor. *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '93*, Copenhagen, Denmark, June 1993. ACM Press.

[31] G. L. Steele Jr. Building interpreters by composing monads. In POPL1994 [28], pages 472–492.

[32] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In Hughes [16], pages 165–191. LNCS 523.