Type Specialisation for the λ-calculus

John Hughes

Department of Computer Science, Chalmers Technical University, S-41296 Göteborg, Sweden. URL: `http://www.cs.chalmers.se/~rjmh`.

1 Introduction

Partial evaluation is a powerful automated strategy for transforming programs, some of whose inputs are known. The classic simple example is the *power* function,

$$power\ n\ x = \textbf{if}\ n = 1\ \textbf{then}\ x\ \textbf{else}\ x \times power\ (n-1)\ x$$

which, given that n is known to be 3, can be transformed into the *specialised* version

$$power_3\ x = x \times (x \times x)$$

The computations on known data (*static* computations) are performed by the partial evaluator once and for all, and in general the resulting *residual program* is considerably more efficient than the original.

Over the last decade partial evaluators have developed from experimental toys into well-engineered tools. But the problem of specialising *typed programs* has never been satisfactorily solved. Straightforward methods produce residual programs that operate on the same types of data as the original program, but this may not be appropriate. For example, where the original program needs a sum type, the residual program may actually only use data lying in one summand. The tagging and untagging operations are then an unnecessary overhead; it would be better to simplify the type to the summand actually used. There are at present no really satisfactory methods for this *type specialisation.*

The problem is particularly acute when the program to be specialised is an interpreter. Interpreters are universal programs which can simulate the behaviour of any other; when an interpreter is specialised to the program P, the residual program is equivalent to P, but is expressed in the language that the partial evaluator processes. It can be considered to be *compiled code* for P. But suppose the interpreter is written in a typed language: then values of every type must be represented by injecting them into one *universal type*, a tagged sum of all the types that can occur. When such an interpreter is specialised, the 'compiled code' produced still operates on tagged values of the universal type, and the performance benefits of compiling a typed language are lost.

Jones calls a partial evaluator *optimal* if the result of specialising a self-interpreter for the language the partial evaluator processes to any program P is not only *equivalent* to P, it is *essentially the same* as P. An optimal partial evaluator can 'remove a complete layer of interpretation'. Partial evaluators for typed languages have not been optimal hitherto, because residual programs contain tagging and untagging operations not present in the programs being compiled.

In this paper we present a partial evaluator for the simply typed λ-calculus which can specialise types, and can thus remove all unnecessary tagging and untagging operations. In particular, one universal type in a self-interpreter can be specialised to an arbitrary type in the residual program. We believe this to be the first time this has been achieved. Our partial evaluator is not optimal either, but this is for minor reasons: we believe an optimal specialiser would now be easy to write.

In the next section we'll give an informal introduction to the partial evaluator via examples. Then we shall specify its behaviour formally via a set of inference rules. We shall go on to describe how the inference system has been implemented, and to discuss an interesting example: specialisation of an interpreter for the typed λ-calculus. Finally we shall sketch future improvements, describe related work, and conclude.

2 Informal Description

Like many other partial evaluators, ours processes a *two-level* language; that is, each construct in the source program is labelled either *static* or *dynamic*, and the partial evaluator performs static computations and builds dynamic ones into the residual program. For example, the number three can be appear either statically (3) or dynamically ($\overline{3}$) — we will consistently mark dynamic constructs by overlining, as in this case. Binding-times (static *vs.* dynamic) are reflected in the types: 3 is of type **int** while $\overline{3}$ is of type $\overline{\mathbf{int}}$.

Every expression has a *static value*: the static value of 3 is of course 3, while the static value of $\overline{3}$ is $\bullet$, which is how we write the unique element of the one-point type **void**. Intuitively this represents the fact that we have no static information about the value of a dynamic expression.

Every expression gives rise to a *residual expression* in the specialised program. The residual expression of $\overline{3}$ is of course 3, while the residual expression of 3 is $\bullet$ — because we need not duplicate at run-time values which were already available during partial evaluation.

Static values of base type can be made dynamic by 'lifting': $\overline{\mathbf{lift}}\ (3+1)$ is equivalent to $\overline{4}$.

Expressions can be *partially static*: for example, $inl\ \overline{3}$ might be an expression of type $\overline{\mathbf{int}} + \overline{\mathbf{bool}}$, a static sum of two dynamic types. The static value of this expression is $inl\ \bullet$: we know statically which summand the value lies in, but we know nothing about the component value. And as promised, the residual expression is *untagged*: it is just 3. It is not necessary to repeat in the residual program the static tag which is known at specialisation time.

Now suppose the program to be specialised contains a dynamic conditional expression such as $\overline{\mathbf{if}}\ x\ \overline{\mathbf{then}}\ inl\ \overline{3}\ \overline{\mathbf{else}}\ inr\ \overline{\mathbf{true}}$. If the conditional were static there would be no difficulty here, but since it is dynamic there are two problems:

- the two arms have *different* static values, namely $inl\ \bullet$ and $inr\ \bullet$, so what is the static value of the whole conditional to be?
- because the conditional is dynamic it will appear in the residual program, but the residual expressions of the two arms, 3 and **true**, have different types; the residual program will be ill-typed.

Previous partial evaluators have forced the result of a dynamic conditional to be completely dynamic, but we see this as a fundamental obstacle to type specialisation. Our specialiser in contrast rejects this program. The two arms of a dynamic conditional *must* have the same static value, so that the static value of the whole expression is well-defined. This in turn guarantees that the residual expressions of the two branches have the same type, and so the residual program is well-typed. We see this as very natural behaviour: just as compilers for typed languages must reject programs which are ill-typed, so partial evaluators (which are used for compilation) must reject programs which are 'ill-static-valued'. But as far as we know, ours is the first partial evaluator to reject some of its inputs.

Another novelty is that we allow dynamic values to contain static components: for example we allow $\mathbf{int} \mathbin{\overline{+}} \mathbf{bool}$, a dynamic sum of static components. Two elements of this type are $\overline{\mathit{inl}}\ 3$ and $\overline{\mathit{inr}}\ \mathbf{true}$. The interpretation is that we do not know at specialisation time which summand a value belongs to, but we know that *if* it belongs to the left summand for example, then its static value is 3. A static value of this type must contain a static value for *each* summand: we write $x + y$ where x and y are the static values of the left and right summands respectively. For example, both expressions above might have static value $3+\mathbf{true}$. As usual, the residual expressions do not duplicate the static information; in this case they are $\mathit{inl}\ \bullet$ and $\mathit{inr}\ \bullet$.

The static value of an expression depends partly on its context. For example, $\overline{\mathit{inl}}\ 3$ could have static value $3+x$ for any x. But in the context $\overline{\mathbf{if}}\ z\ \overline{\mathbf{then}}\ \overline{\mathit{inl}}\ 3\ \overline{\mathbf{else}}\ \overline{\mathit{inr}}\ \mathbf{true}$ its static value must be $3 + \mathbf{true}$. Static values have to be computed by a process more like type inference than evaluation. Again, this is not surprising since static computation is supposed to model type inference among other things.

In order to specialise the body of a λ-expression we need to know the static value of its argument; we therefore take the static values of functions to be *pairs* of the static values of the argument and result. We write these pairs as $x \to y$. For example, in the expression $(\lambda f.f\ 3)(\lambda x.\overline{\mathbf{lift}}(x+1))$ then the λ-expressions have static values $(3 \to \mathbf{void}) \to \mathbf{void}$ and $3 \to \mathbf{void}$ respectively. The residual expression in this case is $(\lambda f.f\ \bullet)(\lambda x.4)$.

Notice that we do not need to unfold static function applications in order to derive their static value: we can infer it anyway. We have exploited this fact to write a partial evaluator which never uses unfolding. We thereby escape the need for renaming of variables. This decision simplifies the partial evaluator somewhat, although of course unfolding is desirable in its own right, and would need to be reintroduced in a tool intended for practical use.

So far we have described a *monovariant* specialiser: each expression in the source gives rise to exactly one residual expression in the result. But this is really very restrictive — for example, the expression $(\lambda f.f\ 3 + f\ 4)(\lambda x.\overline{\mathbf{lift}}(x+1))$ is rejected because we cannot specialise the second λ-expression to two different values at the same time. We solve the problem by introducing *polyvariant* expressions $\mathbf{poly}\ e$, which must be specialised at each use. In the example, we can make the second λ-expression polyvariant by rewriting the example as $(\lambda f.\mathbf{spec}\ f\ 3 + \mathbf{spec}\ f\ 4)(\mathbf{poly}(\lambda x.\overline{\mathbf{lift}}(x+1)))$. The static value of a polyvariant expression is a *tuple* of its static values at each use, and the residual expression is a tuple of corresponding specialisations. Our example is specialised to $(\lambda f.\pi_1 f\ \ \bullet + \pi_2 f\ \ \bullet)(\lambda x.4, \lambda x.5)$.

Finally let us consider the *power* function again. Using **letrec** as syntactic sugar, we can define it as

$$\begin{array}{l}\textbf{letrec}\, power = \textbf{poly}(\lambda n \lambda x.\textbf{if } n = 1 \textbf{ then } x \textbf{ else } x \overline{\times}\, \textbf{spec } power\ (n-1) x) \\ \textbf{in spec } power\ 3\end{array}$$

The *power* function must be polyvariant since it will be specialised to different arguments. The result of specialisation is

$$\begin{array}{l}\textbf{letrec}\, power_3 = \lambda n \lambda x.x \times power_2\ \bullet\ x \\ \qquad power_2 = \lambda n \lambda x.x \times power_1\ \bullet\ x \\ \qquad power_1 = \lambda n \lambda x.x \\ \textbf{in } power\ \bullet\end{array}$$

Our specialiser actually binds *power* to a tuple, but it would be easy to produce multiple definitions as we have shown here. Another important improvement would be to recognise residual parameters of type **void** and omit them. This would be easy to do since the types of residual expressions are of course known at specialisation time. The result in this case would be

$$\begin{array}{l}\textbf{letrec}\, power_3 = \lambda x.x \times power_2\ x \\ \qquad power_2 = \lambda x.x \times power_1\ x \\ \qquad power_1 = \lambda x.x \\ \textbf{in } power_3\end{array}$$

Finally, if our specialiser used unfolding, we would obtain the same residual definition as in the first section of this paper.

3 Specifying the Partial Evaluator

Now that we have introduced the ideas our specialiser is based on we will define it formally. The language we process is a two-level simply typed λ-calculus with the syntax

$$\begin{array}{rl} e ::= & x \mid e\ e \mid \lambda x.e \mid \textbf{fix}\ e \\ \mid & (e,e) \mid \pi_1\ e \mid \pi_2\ e \\ \mid & \mathit{inl}\ e \mid \mathit{inr}\ e \mid \textbf{case}\ e\ \textbf{of}\ \mathit{inl}\ x : e,\ \mathit{inr}\ y : e \\ \mid & \overline{\mathit{inl}}\ e \mid \overline{\mathit{inr}}\ e \mid \overline{\textbf{case}}\ e\ \overline{\textbf{of}}\ \overline{\mathit{inl}}\ x : e,\ \overline{\mathit{inr}}\ y : e \\ \mid & n \mid e + e \mid e - e \mid \ldots \\ \mid & \overline{n} \mid e \overline{+} e \mid e \overline{-} e \mid \ldots \\ \mid & \textbf{poly}\ e \mid \textbf{spec}\ e \\ \mid & \overline{\textbf{lift}}\ e \mid \bullet \end{array}$$

where x and y are variables and n is an integer. The types are given by

$$\begin{array}{rl} \tau ::= & \textbf{void} \mid \textbf{int} \mid \overline{\textbf{int}} \mid \tau \times \tau \mid \tau \to \tau \\ \mid & \tau + \tau \mid \tau \overline{+} \tau \mid \textbf{poly}\ \tau \end{array}$$

Notice that only base and sum types come in static and dynamic versions: we do not need to distinguish static and dynamic functions since we do not unfold them, and similarly static and dynamic pairs are unnecessary.

Types may be recursive, although there is no special syntax for this. Types are simply regular trees conforming to the grammar above.

The typing rules should be obvious and will not be given here: we just point out that polyvariant expressions have types of the form **poly** τ, so the type system forces them to be specialised before use.

We define the *static content* of each type, $|\tau|$, to be the type of the corresponding static values:

$$\begin{array}{rlrl}
|\mathbf{void}| &= \mathbf{void} & |\sigma \to \tau| &= |\sigma| \times |\tau| \\
|\mathbf{int}| &= \mathbf{int} & |\sigma + \tau| &= |\sigma| + |\tau| \\
|\overline{\mathbf{int}}| &= \mathbf{void} & |\sigma \mathbin{\overline{+}} \tau| &= |\sigma| \times |\tau| \\
|\sigma \times \tau| &= |\sigma| \times |\tau| & |\mathbf{poly}\ \tau| &= |\tau|^*
\end{array}$$

The static content of a recursive type is itself a recursive type; this will oblige us to allow infinite static values — in practice cyclic graphs.

We shall write $a : \tau \sim x$ to mean that the value a of type τ *matches* static value x. The semantics of the matching relation is defined by

$$\begin{array}{rcl}
a : \mathbf{void} \sim \bullet &\iff& \mathbf{true} \\
a : \mathbf{int} \sim n &\iff& a = n \\
a : \overline{\mathbf{int}} \sim \bullet &\iff& \mathbf{true} \\
(a, b) : \sigma \times \tau \sim (x, y) &\iff& a : \sigma \sim x \wedge b : \tau \sim y \\
a : \sigma \to \tau \sim x \to y &\iff& \forall b. b : \sigma \sim x \Rightarrow a\ b : \tau \sim y \\
a : \sigma + \tau \sim inl\ x &\iff& \exists b. a = inl\ b \wedge b : \sigma \sim x \\
a : \sigma + \tau \sim inr\ y &\iff& \exists b. a = inr\ b \wedge b : \sigma \sim y \\
a : \sigma \mathbin{\overline{+}} \tau \sim x + y &\iff& \exists b. (a = inl\ b \wedge b : \sigma \sim x) \vee \\
&& \quad (a = inr\ b \wedge b : \tau \sim y) \\
(a_1 \ldots a_k) : \mathbf{poly}\ \tau \sim (x_1 \ldots x_n) &\iff& k = n \wedge \forall i. a_i : \tau \sim x_i
\end{array}$$

The type of residual expressions depends both on their type and on their static value. We write τ/x for the type of residual expressions derived from expressions of type τ with static value x.

$$\begin{array}{rlrl}
\mathbf{void}/\bullet &= \mathbf{void} & (\sigma \to \tau)/(x \to y) &= \sigma/x \to \tau/y \\
\mathbf{int}/n &= \mathbf{void} & (\sigma + \tau)/inl\ x &= \sigma/x \\
\overline{\mathbf{int}}/\bullet &= \mathbf{int} & (\sigma + \tau)/inr\ y &= \tau/y \\
(\sigma \times \tau)/(x, y) &= \sigma/x \times \tau/y & (\sigma \mathbin{\overline{+}} \tau)/(x + y) &= \sigma/x + \tau/y \\
&& \mathbf{poly}\ \tau/(x_1 \ldots x_k) &= \tau/x_1 \times \ldots \times \tau/x_k
\end{array}$$

We will write $e : \tau \sim x \hookrightarrow e'$ to mean that the two-level expression e of type τ can be specialised with respect to the static value x of type $|\tau|$ to yield the residual expression e' of type τ/x.

Now we shall specify the partial evaluator by giving a collection of *inference rules* which can be used to infer specialisation judgements. The form of a judgement will be

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

where Γ is a context consisting of a sequence of assumptions about the static values of variables, of the form $u : \tau \sim x$. The inference rules are given in figure 1. The language of residual expressions is not quite the same as the input language: we use **let** expressions in the specialisation of a static **case**, and n-tuples rather than pairs in the rules *(POLY)* and *(SPEC)*. The meanings should be clear.

The final paper should include a definition of the semantic relationship between source and residual expressions, along with a sketch proof of the soundness of this inference system.

4 Implementing the Partial Evaluator

We have constructed an implementation of the partial evaluator which applies the specialisation rules in much the same way that a type checker applies type inference rules. Just as type inference often produces types containing uninstantiated type variables, so our specialiser often produces specialisations containing uninstantiated *static value variables*. For example, if we specialise $inl\ \overline{3}$ then the result is

$$\vdash inl\ \overline{3} : \overline{\mathbf{int}} + \tau \sim \bullet + \alpha \hookrightarrow 3$$

where α is an uninstantiated static value variable. These variables are usually instantiated by unification when the type containing them is used. Our specialiser is implemented in Haskell, and uses a monad to maintain a state containing the next free variable and the variable instantiations so far, just like a type checker. In order to support cyclic static values we use a graph unification algorithm.

However, the specialisation rules are not as simple to implement as Hindley-Milner type inference rules, because they are not all syntax-directed. In particular, static **case** expressions can be specialised either using *(SCASEL)* or *(SCASER)*, depending on whether the static value of the expression inspected lies in the left or right summand. The implementation specialises this expression first, and if its static value is $inl\ x$ it chooses *(SCASEL)*, and similarly if it is $inr\ y$ then it chooses *(SCASER)*. But unfortunately it is quite possible that the static value is only an uninstantiated variable! In this case we cannot choose which rule to apply until later, when the variable becomes instantiated. To handle this we have implemented a 'demon' mechanism within the monad, which allows a computation to be suspended until a particular variable is instantiated.

When the specialisation of an expression must be delayed like this, we do not know its residual expression at the time that enclosing residual expressions are built. We have solved this problem by introducing a kind of 'forward reference' in the residual code: whenever a specialisation is delayed and a demon created we also create a forward reference label which replaces the unknown residual expression. When the demon eventually runs it creates a binding for the label. Thus the residual program is really constructed in two stages: first specialise, producing a program with forward references, then resolve those references, replacing them with the expressions the demons created. Note that labels cannot be considered as ordinary bound variables, because they do not respect scope: the variables in scope in the expression bound to a label are those in scope at its use.

$$\textit{(SINT)}\quad \Gamma \vdash m : \mathbf{int} \sim m \hookrightarrow \bullet \qquad \textit{(DINT)}\quad \Gamma \vdash \overline{m} : \overline{\mathbf{int}} \sim \bullet \hookrightarrow m$$

$$\textit{(VOID)}\quad \Gamma \vdash \bullet : \mathbf{void} \sim \bullet \hookrightarrow \bullet \qquad \textit{(VAR)}\quad \Gamma, u : \tau \sim x \vdash u : \tau \sim x \hookrightarrow u$$

$$\textit{(LIFT)}\quad \frac{\Gamma \vdash e : \mathbf{int} \sim m \hookrightarrow \bullet}{\Gamma \vdash \overline{\mathbf{lift}}\ e : \overline{\mathbf{int}} \sim \bullet \hookrightarrow m} \qquad \textit{(LAM)}\quad \frac{\Gamma, u : \sigma \sim x \vdash e : \tau \sim y \hookrightarrow e'}{\Gamma \vdash \lambda u.e : \sigma \to \tau \sim x \to y \hookrightarrow \lambda u.e'}$$

$$\textit{(APP)}\quad \frac{\Gamma \vdash e_1 : \sigma \to \tau \sim x \to y \hookrightarrow e_1' \qquad \Gamma \vdash e_2 : \sigma \sim x \hookrightarrow e_2'}{\Gamma \vdash e_1\ e_2 : \tau \sim y \hookrightarrow e_1'\ e_2'}$$

$$\textit{(PAIR)}\quad \frac{\Gamma \vdash e_1 : \tau_1 \sim x_1 \hookrightarrow e_1' \qquad \Gamma \vdash e_2 : \tau_2 \sim x_2 \hookrightarrow e_2'}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2 \sim (x_1, x_2) \hookrightarrow (e_1', e_2')}$$

$$\textit{(FST)}\quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2 \sim (x_1, x_2) \hookrightarrow e'}{\Gamma \vdash \pi_1\ e : \tau_1 \sim x_1 \hookrightarrow \pi_1\ e'} \qquad \textit{(SND)}\quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2 \sim (x_1, x_2) \hookrightarrow e'}{\Gamma \vdash \pi_2\ e : \tau_2 \sim x_2 \hookrightarrow \pi_2\ e'}$$

$$\textit{(SINL)}\quad \frac{\Gamma \vdash e : \tau \sim x \hookrightarrow e'}{\Gamma \vdash \mathit{inl}\, e : \tau + \sigma \sim \mathit{inl}\, x \hookrightarrow e'} \qquad \textit{(SINR)}\quad \frac{\Gamma \vdash e : \tau \sim x \hookrightarrow e'}{\Gamma \vdash \mathit{inr}\, e : \sigma + \tau \sim \mathit{inr}\, x \hookrightarrow e'}$$

$$\textit{(SCASEL)}\quad \frac{\Gamma \vdash e_0 : \sigma + \tau \sim \mathit{inl}\, x \hookrightarrow e_0' \qquad \Gamma, u_1 : \sigma \sim x \vdash e_1 : \rho \sim y \hookrightarrow e_1'}{\Gamma \vdash \mathbf{case}\ e_0\ \mathbf{of}\ \mathit{inl}\, u_1 : e_1,\ \mathit{inr}\, u_2 : e_2 : \rho \sim y \hookrightarrow \mathbf{let}\ u_1 = e_0'\ \mathbf{in}\ e_1'}$$

$$\textit{(SCASER)}\quad \frac{\Gamma \vdash e_0 : \sigma + \tau \sim \mathit{inr}\, x \hookrightarrow e_0' \qquad \Gamma, u_2 : \tau \sim x \vdash e_2 : \rho \sim y \hookrightarrow e_2'}{\Gamma \vdash \mathbf{case}\ e_0\ \mathbf{of}\ \mathit{inl}\, u_1 : e_1,\ \mathit{inr}\, u_2 : e_2 : \rho \sim y \hookrightarrow \mathbf{let}\ u_2 = e_0'\ \mathbf{in}\ e_2'}$$

$$\textit{(DINL)}\quad \frac{\Gamma \vdash e : \sigma \sim x \hookrightarrow e' \qquad y : |\tau|}{\Gamma \vdash \overline{\mathit{inl}}\, e : \sigma \overline{+} \tau \sim x + y \hookrightarrow \mathit{inl}\, e'} \qquad \textit{(DINR)}\quad \frac{\Gamma \vdash e : \tau \sim y \hookrightarrow e' \qquad x : |\sigma|}{\Gamma \vdash \overline{\mathit{inr}}\, e : \sigma \overline{+} \tau \sim x + y \hookrightarrow \mathit{inr}\, e'}$$

$$\textit{(DCASE)}\quad \frac{\Gamma \vdash e_0 : \sigma \overline{+} \tau \sim x + y \hookrightarrow e_0' \qquad \begin{array}{l} \Gamma, u : \sigma \sim x \vdash e_1 : \rho \sim z \hookrightarrow e_1' \\ \Gamma, v : \tau \sim y \vdash e_2 : \rho \sim z \hookrightarrow e_2' \end{array}}{\Gamma \vdash \overline{\mathbf{case}}\ e_0\ \overline{\mathbf{of}}\ \overline{\mathit{inl}}\, u : e_1,\ \overline{\mathit{inr}}\, v : e_2 : \rho \sim z \hookrightarrow \mathbf{case}\ e_0'\ \mathbf{of}\ \mathit{inl}\, u : e_1',\ \mathit{inr}\, v : e_2'}$$

$$\textit{(SOP)}\quad \frac{\Gamma \vdash e_1 : \mathbf{int} \sim m \hookrightarrow \bullet \qquad \Gamma \vdash e_2 : \mathbf{int} \sim n \hookrightarrow \bullet}{\Gamma \vdash e_1 \oplus e_2 : \mathbf{int} \sim m \oplus n \hookrightarrow \bullet}$$

$$\textit{(DOP)}\quad \frac{\Gamma \vdash e_1 : \overline{\mathbf{int}} \sim \bullet \hookrightarrow e_1' \qquad \Gamma \vdash e_2 : \overline{\mathbf{int}} \sim \bullet \hookrightarrow e_2'}{\Gamma \vdash e_1 \overline{\oplus} e_2 : \overline{\mathbf{int}} \sim \bullet \hookrightarrow e_1' \oplus e_2'}$$

$$\textit{(FIX)}\quad \frac{\Gamma \vdash e : \tau \to \tau \sim x \to x \hookrightarrow e'}{\Gamma \vdash \mathbf{fix}\ e : \tau \sim x \hookrightarrow \mathbf{fix}\ e'}$$

$$\textit{(POLY)}\quad \frac{\Gamma \vdash e : \tau \sim x_1 \hookrightarrow e_1 \ldots \Gamma \vdash e : \tau \sim x_k \hookrightarrow e_k}{\Gamma \vdash \mathbf{poly}\ e : \mathbf{poly}\ \tau \sim (x_1 \ldots x_k) \hookrightarrow (e_1 \ldots e_k)}$$

$$\textit{(SPEC)}\quad \frac{\Gamma \vdash e : \mathbf{poly}\ \tau \sim (x_1 \ldots x_i \ldots x_k) \hookrightarrow e'}{\Gamma \vdash \mathbf{spec}\ e : \tau \sim x_i \hookrightarrow \pi_i\ e'}$$

Fig. 1. The Specialisation Rules.

It is possible that some references are never resolved, if the corresponding demon is waiting for a variable that is never instantiated. For example, if we try to specialise $\lambda x.\mathbf{case}\ x\ \mathbf{of}\ \mathit{inl}\ a : \overline{3},\ \mathit{inr}\ b : \overline{4}$ the residual expression will be $\lambda x.\nu_1$ where ν_1 is an unresolved forward reference — a demon is created to wait for the static value of x, but it is never run. We consider such inputs to be erroneous: after all, partial evaluation presupposes that all static inputs are given, and that is not true in this case.

The other rules which are not syntax-directed are *(POLY)* and *(SPEC)*: in *(POLY)* we do not know which variants to create, and in *(SPEC)* we do not know which variant to choose. In the implementation we have simplified the problem by replacing the n-tuples by nested pairs. Each inference rule then splits into two:

$$\textit{(POLY1)}\quad \frac{\Gamma \vdash e : \tau \sim x_1 \hookrightarrow e_1 \quad \Gamma \vdash \mathbf{poly}\ e : \mathbf{poly}\ \tau \sim x_2 \hookrightarrow e_2}{\Gamma \vdash \mathbf{poly}\ e : \mathbf{poly}\ \tau \sim (x_1, x_2) \hookrightarrow (e_1, e_2)}$$

$$\textit{(POLY2)}\quad \Gamma \vdash \mathbf{poly}\ e : \mathbf{poly}\ \tau \sim \bullet \hookrightarrow \bullet$$

$$\textit{(SPEC1)}\quad \frac{\Gamma \vdash e : \mathbf{poly}\ \tau \sim (x_1, x_2) \hookrightarrow e'}{\Gamma \vdash \mathbf{spec}\ e : \tau \sim x_1 \hookrightarrow \pi_1\ e'}$$

$$\textit{(SPEC2)}\quad \frac{\Gamma \vdash e : \mathbf{poly}\ \tau \sim (x_1, x_2) \hookrightarrow e'}{\Gamma \vdash e : \mathbf{poly}\ \tau \sim x_2 \hookrightarrow \pi_2\ e'}$$

Polyvariant expressions are specialised by creating a demon which waits for the static value to be instantiated: when it is instantiated to a pair then rule *(POLY1)* is applied to create an appropriate specialisation to the first component of the pair, and another demon is created waiting for the second component to be instantiated. If such a variable remains uninstantiated at the end of specialisation, then rule *(POLY2)* is applied to instantiate it to $\bullet$. In other words, we create as many specialisations as are required, then set the 'tail of the list' to $\bullet$.

It is the two *(SPECi)* rules that actually instantiate these variables, driving the polyvariant specialisation. If we try to specialise a **spec** e expression while the static value of e is still uninstantiated, then we apply rule *(SPEC1)*, thereby instantiating the static value of e to a pair, running the demon, and installing this specialisation as the first in the 'n-tuple'. Since the order of the elements of this tuple is really irrelevant, it is not necessary to consider applying *(SPEC2)* in this situation.

If on the other hand the static value of e is already a pair (x_1, x_2), then we do not know whether or not x_1 is the right specialisation to use at this point. Moreover we cannot find out except by trial and error. We therefore apply rule *(SPEC1)*, but if further specialisation fails (via a unification failure), we backtrack to this point and apply *(SPEC2)* instead. By repeated application of *(SPEC2)* followed by *(SPEC1)* we try each previously generated specialisation, and if none works, we generate a new one. Our monad therefore needs to support backtracking in addition to maintaining variables, forward references and demons.

The efficiency of the specialiser depends critically on how quickly the wrong alternatives in this process lead to failure. This in turn depends on which order the rules are applied in. An early version specialised applications by first specialising the function, and then specialising the argument. As a result the argument's static value was not known at the time the function itself was specialised, and most of the real work had to be delayed via a demon until after the argument was specialised. When the function itself was a **spec** expression, this meant that *every* specialisation matched at first, leading to very poor performance. Reversing the order to specialise the argument first meant that functions were usually specialised to *known* static values, and in the case of a **spec** expression the wrong alternatives could usually be eliminated at once. Applying the *(SPECi)* rules then closely resembles searching a pending list for matching arguments. This change in the order cut the specialisation time for one simple example from over 27 seconds to around a half a second: it is important!

5 Specialising an Interpreter

The acid test for our partial evaluator is whether it can remove an entire layer of interpretation, in the sense of eliminating type tags. To demonstrate that it can, we specialise a very small interpreter for the simply typed λ-calculus with constants. Because our partial evaluator does not handle strings, we represent variable names by integers — but these are just used as distinct names, not deBruin numbers. In a slightly extended language, the interpreter is given below.

```
$\mathbf{data}\ E = Cn\ \mathbf{int} \mid Vr\ \mathbf{int} \mid Lm\ \mathbf{int}\ E \mid Ap\ E\ E$
$\mathbf{data}\ U = Num\ \overline{\mathbf{int}} \mid Fun\ (U \to U) \mid Wrong$
$\mathbf{type}\ Env = \mathbf{poly}\ (\mathbf{int} \to U)$

$\mathbf{letrec}\, eval = \mathbf{poly}\ \lambda env \lambda e.$
          $\mathbf{case}\ e\ \mathbf{of}$
            $Cn\ n$      $: Num\ (\overline{\mathbf{lift}}\ n)$
            $Vr\ i$      $: \mathbf{spec}\ env\ i$
            $Lm\ i\ e$   $: Fun\ (\lambda v.\mathbf{spec}\ eval$
                                $(\mathbf{poly}\ \lambda j.\mathbf{if}\ i = j\ \mathbf{then}\ v\ \mathbf{else}\ \mathbf{spec}\ env\ j)$
                                $e$
            $Ap\ e_1\ e_2$ $: \mathbf{case}\ \mathbf{spec}\ eval\ env\ e_1\ \mathbf{of}$
                         $Num\ k : Wrong$
                         $Fun\ f\ \ : f\ (\mathbf{spec}\ eval\ env\ e_2)$
                         $Wrong : Wrong$
$\mathbf{in}\ \mathbf{spec}\ eval\ (\mathbf{poly}\ \lambda i.\, Wrong)$
```

The value of this expression is a partial application of *eval* to the initial environment, that is, a function from closed expressions to their values.

Here the type of expressions E is purely static, so expression values should disappear entirely in residual programs, while the universal type U is a *static* sum of products, and so its residual types should be derived from just one of the summands. Any type involving **int** and function arrow can be derived as a residual of U! For example, one element of $|U|$ is $Fun\ (Num\ \bullet \to Num\ \bullet)$. The corresponding residual type is just $\mathbf{int} \to \mathbf{int}$. Similarly any other type, including recursive types, can be derived as a residual with respect to a suitable static value.

The *eval* function is of course polyvariant — it must be specialised to many different expressions. We also make the *environment* polyvariant, because it will be 'specialised' to look up many different variables. For example, if the environment is specialised to look up variables 1 and 2, then its static value will be a 2-tuple (x_1, x_2) whose components are the static values of the two variables, and its residual expression will be a tuple (e_1, e_2) of the two values.

To specialise this interpreter we must desugar it into the language our partial evaluator accepts, and apply it to a suitable input. The desugaring consists of replacing the constructors of E and U by suitable compositions of *inl* and *inr*, desugaring **case** expressions similarly, replacing **if** by **case**, and replacing **letrec** by **fix**.

We have specialised our interpreter to a number of tiny expressions. The simplest is $(Cn\ 3)$. The static value in this case (resugared) is $Num\ \bullet$, indicating that the type is **int**, and the residual code is $\pi_1\ (\mathbf{fix}\ \lambda eval.(\lambda env.\lambda e.\mathbf{let}\ n = e\ \mathbf{in}\ 3, \bullet))\ \bullet\ \bullet$. This code is not resugared in any way, which is painfully obvious! But it can easily be simplified. All of *env*, *e*, *n* and the second component of the **fix** are of type **void**; if values of type **void** are elided the result is $\mathbf{fix}\ \lambda eval.3$! Resugaring the **letrec**, we write the result as

$$\begin{array}{l}\mathbf{letrec}\, eval = 3 \\ \mathbf{in}\ eval\end{array}$$

We will resugar the remaining examples in this way by hand. Future implementations will perform this simplification automatically.

Our next example is $(Vr\ 1)$, an input which is actually incorrect — the variable is not bound. The static value is *Wrong* (the result of applying the initial environment to 1), and the resugared residual code is

$$\begin{array}{l}\mathbf{letrec}\, eval = \bullet \\ \mathbf{in}\ eval\end{array}$$

Taking a slightly larger example, $Ap\ (Lm\ 1\ (Vr\ 1))\ (Cn\ 3)$, the static value is again $Num\ \bullet$ because the type is **int**, and the residual code is

$$\begin{array}{ll}\mathbf{letrec} & eval_1 = \mathbf{let}\ f = eval_2\ \mathbf{in}\ f\ eval_3 \\ & eval_2 = \lambda v.eval_4\ v \\ & eval_3 = 3 \\ & eval_4 = \lambda env.env \\ \mathbf{in}\ eval_1 & \end{array}$$

Here the **let**-expression is the residual from the static **case** that checked that f was a function; $eval_2$ is the compiled λ-expression; $eval_4$ is the compiled occurrence of $(Vr\ 1)$, a function from the environment (which is a single value) to the value of the variable; $eval_3$ is the compiled constant. The code is much larger than that produced by other partial evaluators, but this is largely because we do not use unfolding. If we unfold calls of *eval* the resulting code is

$$\mathbf{let}\ f = \lambda v.v\ \mathbf{in}\ f\ 3$$

It would not be difficult to add unfolding to the implementation, and obviously we should.

As a final example, involving a higher-order function, consider

$$Ap\ (Lm\ 1\ (Ap\ (Vr\ 1)\ (Cn\ 3)))\ (Lm\ 2\ (Vr\ 2))$$

The sugared residual code is

$$\begin{aligned}
\textbf{letrec}\ & eval_1 = \textbf{let}\ f = eval_2\ \textbf{in}\ f\ eval_3 \\
& eval_2 = \lambda v.eval_4\ v \\
& eval_3 = \lambda v.eval_5\ v \\
& eval_4 = \lambda env.\textbf{let}\ f = eval_6\ env\ \textbf{in}\ f\ (eval_7\ env) \\
& eval_5 = \lambda env.env \\
& eval_6 = \lambda env.env \\
& eval_7 = \lambda env.3 \\
\textbf{in}\ & eval_1
\end{aligned}$$

The pattern should now be clear. Each expression is compiled as a residual version of *eval*. Expressions with free variables are compiled as functions of *env* — those without free variables take a **void** environment which is erased. Applications are compiled as a **let**, λ-expressions as $\lambda v.\ldots$, variables as projection functions on the environment, and constants as themselves. And the important point: no run-time type tags appear!

After unfolding calls of *eval* the residual code becomes

$$\textbf{let}\ f = \lambda v.(\textbf{let}\ f = v\ \textbf{in}\ f\ 3)\ \textbf{in}\ f\ (\lambda v.v)$$

which is very close to optimal. Unfolding linear **let**s also would indeed yield an optimal result.

6 Improvements and Further Work

We have already mentioned a number of desirable improvements: extension of the source language, **void** erasure, unfolding, the addition of user-defined types and constructors. One fairly easy extension would be to generate explicit definitions of residual types from the graph of static values. This would be necessary to process languages such as ML and Haskell, which require that recursive types be explicitly defined.

One serious deficiency is that residual programs are always simply typed. It would therefore be impossible to specialise a polymorphic language optimally using only these techniques. It is not obvious how to make an extension to polymorphism: the standard Milner polymorphic type inference relies on generalising type variables which remain uninstantiated after an expression's type is inferred — but in our case it is hard to tell when inference of an expression's static value is complete. There could be a demon waiting to instantiate more variables.

Our **poly** construct gives us only crude control over polyvariance. For example, in an interpreter for a simply typed language the *eval* function must of course be polyvariant so that it can be specialised to many different expressions. But our specialiser will then be happy to generate *many* specialisations of *eval* to the *same* expression with different static environments! In particular, the body of a λ-expression might thus be compiled several times, with different types for its free variables. This is firstly not the intention if the object language is simply-typed, and secondly may lead to non-termination on ill-typed inputs. We would like to specify that *eval* is polyvariant in its first argument, but that for each value of the first argument there should be only one specialisation to the others!

One idea would be to introduce a *polyvariant function* type, whose static content would be a tuple of pairs with *different* first components. But it is unclear at present how to extend our implementation to handle these.

It would be desirable to link a binding-time analysis to our partial evaluator. We believe this would be very simple because the type discipline of our two-level source language is just that of the simply-typed λ-calculus: there are no awkward dependencies between the binding-time of the test in a conditional and its result.

In the longer term we hope to achieve self-application of the partial evaluator. Doing so will require a careful redesign of the monad to separate meta-static and static variables. The residual programs will of course still use a similar monad, with all the mechanism of demons and uninstantiated variables. Generated compilers will pass all data around via unification: reasonable enough for types, but a little curious for the program being compiled! It may be possible to replace unification by a simpler, more functional mechanism, in some cases.

7 Related Work

The first partial evaluator for the λ-calculus, Gomard and Jones' λMIX, also processes a two-level typed language [GJ91]. But in contrast to ours, there is only one dynamic type: **code**. Consequently residual programs are untyped. One benefit is that interpreters to be specialised by λMIX need not inject dynamic values into a universal type: no explicit type tags appear in either source or residual programs. On the other hand, the type tags are actually present in the implementation of the dynamically typed language, and there is no way to get rid of them.

Restricted forms of type specialisation have been used in the past. Romanenko's *arity raising* replaces a list whose length is known statically by a tuple [Rom90]. Launchbury generalised the idea using *projections* to divide data into a static structure containing dynamic components [Lau91]. Such a partially static structure can be replaced in the residual program by a tuple of its dynamic components. However, Launchbury's partial evaluator cannot remove type tags in general because the result of a dynamic conditional is forced to be purely dynamic, and so type tags become unknown at that point. Similarly the dynamic components cannot contain nested static parts, so their types cannot be specialised. Finally, this technique is limited to first-order programs.

Weise and Ruf describe an online method for computing the static parts of dynamic values in an untyped language [WR90]. They can even compute static parts of dynamic conditionals, by 'generalising' the static parts of the two branches. In the case that they match, the dynamic conditional has an informative static value, and in the case that they do not, it doesn't. They can handle function values by treating them as closures. However, they cannot represent disjunctive static information as we do using dynamic sums, and they do not use the information obtained to change the representations of dynamic values.

Mogensen has proposed a form of type specialisation called *constructor specialisation* [Mog93]. Here some components of a user-defined algebraic type may be classified as static. For example, consider the type of integer lists

$$\textbf{data intlist} = \mathit{Nil} \mid \mathit{Cons}\ \textbf{int intlist}$$

Suppose that the integer elements are static. Then applications of *Cons* are *specialised* to their integer parameters; if *Cons* is applied to 1, 2 and 3, then specialised constructors Cons_1, Cons_2 and Cons_3 are defined. The residual program will then contain a specialised type definition

$$\textbf{data intlist}_0 = \mathit{Nil} \mid \mathit{Cons}_1\ \textbf{intlist}_0 \mid \mathit{Cons}_2\ \textbf{intlist}_0 \mid \mathit{Cons}_3\ \textbf{intlist}_0$$

instead of the original. All **case** expressions matching on *Cons* must of course also be specialised, to have a case for each new constructor. In each such case, the value of the static component is known, and so a static value has in effect been extracted from dynamic data.

However, constructor specialisation cannot remove type tags from residual programs — only specialise them. The specialisation of types is monovariant: each type in the source yields one type in the residual program. The method is limited to first order programs. But like us, Mogensen uses 'forward references' to generate the residual program out-of-order.

It is interesting to note that our specialiser does not achieve constructor specialisation. Given the type above, our specialiser would insist that all applications of *Cons* that could reach the same point had the *same* static component. Our specialiser is polyvariant for types, but monovariant for constructors — it is intriguing to wonder whether constructor specialisation could be incorporated.

Constructor specialisation has been generalised by Dussart et al. [DBV95] to be polyvariant: one data type in the source program may give rise to arbitrarily many in the residual program. Like ours, their partial evaluator collects static information about dynamic expressions which is used to decide their residual type. In this case the static information is expressed as a grammar, which should be compared to our potentially cyclic static values. There is clearly a close relationship to our own work — but also significant differences. The static information associated with dynamic expressions is determined by abstract interpretation, not by inference as in our case. This analysis precedes specialisation, rather than being an integral part of it. It requires approximations to improve termination, such as a restriction to so-called 'flat grammars'. In the residual programs, datatypes with constructors are transformed into datatypes with (other) constructors: in other words, tags are never eliminated altogether. Finally, the method described is for first order programs only, and a planned extension to higher-order requires a control-flow analysis with attendant approximations. In contrast our inference based approach handles full λ-calculus simply and naturally.

Danvy's recent work on *type-directed partial evaluation* may perhaps be related [Dan96]. Danvy 'residualises' two-level λ-terms given their type. His residual programs are typed, and it is possible to derive different terms from the same source term by residualising it at different types. But it does not seem as though his method solves the problem of type tags in residual programs.

Both λMIX and Danvy's recent work share a curious characteristic: they are in a sense monovariant specialisers. Both use unfolding, and can duplicate expressions in the process, but neither can duplicate expressions without unfolding. So for example, a single recursive function cannot be specialised to two mutually recursive residual functions (unfolding would be dangerous in this case). Polyvariant specialisation can be awkward in a higher-order language because recognising that two specialisations are the same may require comparing static function values. In our work, the static content of a function is a pair, not a function, and so comparisons are unproblematic.

8 Summary and Conclusions

We have presented a new approach to partial evaluation in which the strict one-to-one relationship between expressions and their static values is broken: rather a one-to-many matching relation holds. 'The' static value of an expression cannot be computed, therefore, and instead we *infer* an appropriate value by a process similar to type inference. Using inference rather than computation enables us to associate static information also with dynamic values. This static information is used to transform the type of residual expressions, and so one expression in the source can yield residual expressions with many types when specialised with respect to different static values. A suitable universal type can be specialised to yield any type at all in the residual program, removing the limitation on residual programs that previous partial evaluators have suffered from. With some small extensions we believe our partial evaluator will solve a long-standing open problem: optimal specialisation of a typed language.

However, our work so far is exploratory and the approach raises many questions. We have demonstrated the principle, but there is considerable scope for refinement.

References

[Dan96] Olivier Danvy. Type-directed partial evaluation. In *Symposium on Principles of Programming Languages*. ACM, jan 1996.

[DBV95] Dirk Dussart, Eddy Bevers, and Karel De Vlaminck. Polyvariant Constructor Specialisation. In *Proc. ACM Conference on Partial Evaluation and Program Manipulation*, La Jolla, California, 1995.

[GJ91] C. K. Gomard and N. D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–70, January 1991.

[Lau91] J. Launchbury. *Projection Factorisations in Partial Evaluation (PhD thesis)*, volume 1 of *Distinguished Dissertations in Computer Science*. Cambridge University Press, 1991.

[Mog93] Torben Æ. Mogensen. Constructor specialization. In David Schmidt, editor, *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 22–32, June 1993.

[Rom90] S. A. Romanenko. Arity raiser and its use in program specialisation. In *Proc. 3rd European Symposium on Programming, Lecture Notes in Computer Science Vol. 432*, pages 341–360. Springer-Verlag, May 1990.

[WR90] Daniel Weise and Erik Ruf. Computing types during program specialization. Technical Report CSL-TR-90-441, Stanford Computer Science Laboratory, October 1990.