Type Specialisation for the λ-calculus; or, A New Paradigm for Partial Evaluation based on Type Inference

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,

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

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. Such *type specialisation* was identified as a 'challenging problem in partial evaluation' by Neil Jones in 1987, but there are still no really satisfactory methods for doing it.

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'. Most 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.

Removing these tags carries a risk: there is a possibility that residual programs may become ill typed, in the case where a tag check would have failed. Residual programs therefore need to be type-checked. Rather than leaving this for a post-processor, we have taken it as inspiration for a new kind of partial evaluator: whereas previous ones have been, in a sense, generalised evaluators, ours is a kind of generalised type-checker. In this sense our work introduces a new paradigm for partial evaluation.

The partial evaluator we describe is the first optimal partial evaluator for the simply typed λ-calculus. It can specialise types, and can 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.

In the next section we informally introduce the basic ideas underlying our partial evaluator. Then we specify its behaviour formally via a set of inference rules. We go on to briefly describe the binding-time checker we use before specialisation, and a post-processor that removes trivial residual computations. Next we describe how the specialiser's inference system has been implemented, and discuss an interesting example: specialisation of an interpreter for the typed λ-calculus. Finally we discuss future improvements, describe related work, and conclude.

2 An Informal Introduction

We shall begin in this section by introducing some of the basic concepts underlying our partial evaluator, and explaining why partial evaluation by type inference is interesting.

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 appear either statically (3) or dynamically ($\underline{3}$) — we will consistently mark dynamic constructs by underlining, as in this case. Binding times (static *vs.* dynamic) are reflected in the types: 3 is of type **int** while $\underline{3}$ is of type $\underline{\mathbf{int}}$.

Every expression gives rise to a *residual expression* in the specialised program. The residual expression of $\underline{3}$ is of course 3, while the residual expression of 3 is $\bullet$, which is how we write the unique element of the one-point type[1]. Intuitively, since 3 is known during partial evaluation we can replace it by a dummy value in the specialised program.

We shall use the notation $a \hookrightarrow b$ to mean that source expression a is specialised to residual expression b, for example $\underline{3} \hookrightarrow 3$ and $3 \hookrightarrow \bullet$. But since we are actually

[1] The only element of the one-point type is of course $\perp$, the undefined element, but we prefer to write $\bullet$ to make clear that we mean the dummy value, not the bottom element of some other type. We assume a lazy semantics so that $\bullet$ can be freely passed as a parameter, and so on. If one prefers a strict semantics one must take $\bullet$ to be the defined element of a two-point type instead.

interested in specialising typed programs, we will annotate both source and residual expressions with their types, thus:

$$a : \sigma \hookrightarrow b : \tau$$

We will call σ the *source type* and τ the *residual type*.

The fundamental new idea in this paper is to *propagate static information via residual types*. For example, when we specialise a static integer the residual expression is a dummy value, but the residual type tells us which static value it represents:

$$3 : \mathbf{int} \hookrightarrow \bullet : 3$$

Here '3' is a type with only one element, namely $\bullet$, and which therefore has just the same elements as 4, 5, 6 *etc.* — but which carries different static information.

All expressions, even 'dynamic' ones, carry static information in the form of a residual type. For purely dynamic expressions this is just an ordinary type, for example

$$\underline{3} : \underline{\mathbf{int}} \hookrightarrow 3 : \mathbf{int}$$

But we can often express useful 'partially static' information via a residual type. For example:

- $2 \times \mathbf{int}$, the type of pairs whose first component is statically 2,
- $\mathbf{int} \to (2 \times \mathbf{int})$, the type of functions whose result is a pair with first component statically 2.

This static information is then propagated by type inference.

Our ability to associate static information with dynamic values can lead to very strong specialisation. For example, consider the expression

$$(\underline{\lambda} f.\mathbf{lift}(f \mathbin{\underline{@}} 3)) \mathbin{\underline{@}} (\underline{\lambda} x.x + 1)$$

Here expressions of the form $\underline{\lambda} x.e$ are dynamic λ-expressions, which are transformed into residual λ-expressions in the specialised program. The '$\underline{@}$' operator is dynamic function application, and the **lift** operator converts a static integer into a dynamic one.

To specialise this expression, we infer the residual type of each subexpression. Note that f is applied to an argument with residual type 3, and so must have residual type $3 \to \tau$ for some τ. But then x must have residual type 3, and so $x + 1$ has residual type 4, and $\underline{\lambda} x.x+1$ has residual type $3 \to 4$. This must also be the type of f, and so $f \mathbin{\underline{@}} 3$ has residual type 4. The **lift** operation can now be specialised to 4. The final result is

$$(\lambda f.4)(\lambda x.\bullet) : \mathbf{int}$$

in which the static computations have been removed, and only the dynamic operations and the result of **lift** remain.

Traditional partial evaluators propagate static information in a different way: static expressions are reduced to values, which are then bound to static variables and so on. There is thus a tight connection between unfolding and staticness. For example, Gomard and Jones λ-MIX (the first partial evaluator for the λ-calculus) insists that since dynamic λ-expressions are not unfolded, both their argument and result must also be dynamic. In our example that would force x to be dynamic, and consequently the addition could not be performed by the specialiser. To specialise this example using λ-MIX, one must annotate the λ-expressions static so that the specialiser can unfold them.

In contrast, there is no need to unfold a λ-expression in order to infer its residual type, and our partial evaluator can therefore specialise the example as it stands. But does this really matter — surely unfolding is desirable in its own right? Our answer is: yes it does matter! No realistic partial evaluator can unfold *all* function calls, and it is the calls that are not unfolded that lead to loss of static information. Type inference gives better static information flow than unfolding, which leads to stronger specialisation. In particular, it is the key to optimal specialisation of typed programming languages.

We can also see from this simple example that the residual programs that our specialiser produces tend to manipulate dummy values — they are the spoor left by purely static computations. To achieve optimal specialisation, there must be *no* trace of static computations in residual programs. But fortunately, such useless expressions are easily identified. We remove them in a post-processing phase we call *void erasure*. The residual expression we derived above was

$$(\lambda f.4)(\lambda x.\bullet)$$

Here the entire second λ-expression is useless, and so void erasure removes both it and the variable (f) it is bound to, resulting in just the expression 4.

3 Specifying the Partial Evaluator

Just as a type checker can be concisely specified by a set of type inference rules, so our partial evaluator can be specified by a set of *specialisation rules*. These rules prescribe how to infer *specialisation judgements*, which are just like typing judgements except that, since we are specifying a program transformation, each judgement contains *two* terms and *two* types. The form of a specialisation judgement is

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

Γ is a context containing assumptions on variables, which have a similar form:

$$x : \tau \hookrightarrow e : \tau'$$

where x is the variable and e an expression to substitute for it.

A nice consequence of this approach is that we can specify the partial evaluator in a very modular way. A simple version can be specified with relatively few rules, and each new feature can then be expressed via a new type along with its introduction and elimination rules.

Constructing an implementation of the rules requires a certain amount of cleverness, because they are not all syntax-directed. But we will return to this in a later section: for now, we are just concerned with specifying how the partial evaluator should behave.

3.1 Base Types

We have already seen examples of how base types are specialised. Static and dynamic integer constants are specialised via the rules

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

Primitive operators come in static and dynamic variants, with rules for each operator $\oplus$ of the form

$$\textit{(S}\oplus\textit{)} \quad \frac{\Gamma \vdash e_1 : \mathbf{int} \hookrightarrow e_1' : m_1 \quad \Gamma \vdash e_2 : \mathbf{int} \hookrightarrow e_2' : m_2 \quad m_1 \oplus m_2 = n}{\Gamma \vdash e_1 \oplus e_2 : \mathbf{int} \hookrightarrow \bullet : n}$$

$$\textit{(D}\oplus\textit{)} \quad \frac{\Gamma \vdash e_1 : \underline{\mathbf{int}} \hookrightarrow e_1' : \mathbf{int} \quad \Gamma \vdash e_2 : \underline{\mathbf{int}} \hookrightarrow e_2' : \mathbf{int}}{\Gamma \vdash e_1 \underline{\oplus} e_2 : \underline{\mathbf{int}} \hookrightarrow e_1' \oplus e_2' : \mathbf{int}}$$

The **lift** operation just introduces a suitable constant in the residual program:

$$\textit{(LIFT)} \quad \frac{\Gamma \vdash e : \mathbf{int} \hookrightarrow e' : m}{\Gamma \vdash \mathbf{lift}\ e : \underline{\mathbf{int}} \hookrightarrow m : \mathbf{int}}$$

Clearly, other base types could be added with similar rules. We will just remark that it is often useful to have a **void** type, with one element — namely $\bullet$.

$$\textit{(VOID)} \quad \Gamma \vdash \bullet : \mathbf{void} \hookrightarrow \bullet : \mathbf{void}$$

3.2 Variables and Lets

The context Γ binds variables to a residual expression to be substituted for them: this is how unfolding is implemented. The rule for variables just does the substitution.

$$\textit{(VAR)} \quad \Gamma, x : \tau \hookrightarrow e : \tau' \vdash x : \tau \hookrightarrow e : \tau'$$

Variables can be bound by **let** expressions, which come in two variants, static and dynamic. Static **let** expressions are unfolded by the partial evaluator, whereas dynamic ones create a **let** in the residual program.

$$\textit{(SLET)} \quad \frac{\Gamma \vdash e_1 : \sigma \hookrightarrow e_1' : \sigma' \quad \Gamma, x : \sigma \hookrightarrow e_1' : \sigma' \vdash e_2 : \tau \hookrightarrow e_2' : \tau'}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau \hookrightarrow e_2' : \tau'}$$

$$\textit{(DLET)} \quad \frac{\Gamma \vdash e_1 : \sigma \hookrightarrow e_1' : \sigma' \quad \Gamma, x : \sigma \hookrightarrow x' : \sigma' \vdash e_2 : \tau \hookrightarrow e_2' : \tau'}{\Gamma \vdash \underline{\mathbf{let}}\ x = e_1\ \underline{\mathbf{in}}\ e_2 : \tau \hookrightarrow \mathbf{let}\ x' = e_1'\ \mathbf{in}\ e_2' : \tau'} \quad x' \text{ fresh}$$

Notice that the *only* difference between these two rules is the degree of unfolding in the residual program. In particular, there is no need for a variable bound by a dynamic **<u>let</u>** to itself be dynamic. We have exactly the same 'static information' about x in each case, expressed as its residual type σ'. Only the expression to substitute for x varies.

Traditional partial evaluators bind variables to replacement expressions, but have no concept of residual type. A static variable is bound to an expression which is constant. It follows that a variable bound in a dynamic **<u>let</u>**, which will be replaced by a fresh variable in the residual program, cannot be static. This case shows clearly how a type-inference-based partial evaluator can achieve stronger results than an unfolding-based one. [2]

3.3 Product Types

We introduce a dynamic product type, which generates pairs in the residual program. For the time being, we will ignore the possibility of static products, and so for type-setting reasons we refrain from underlining the brackets and comma of a pair.

$$\textit{(DPAIR)} \quad \frac{\Gamma \vdash e_1 : \sigma \hookrightarrow e_1' : \sigma' \quad \Gamma \vdash e_2 : \tau \hookrightarrow e_2' : \tau'}{\Gamma \vdash (e_1, e_2) : \sigma \underline{\times} \tau \hookrightarrow (e_1', e_2') : \sigma' \times \tau'}$$

$$\textit{(DFST)} \quad \frac{\Gamma \vdash e : \sigma \underline{\times} \tau \hookrightarrow e' : \sigma' \times \tau'}{\Gamma \vdash \underline{\pi}_1\ e : \sigma \hookrightarrow \pi_1\ e' : \sigma'} \qquad \textit{(DSND)} \quad \frac{\Gamma \vdash e : \sigma \underline{\times} \tau \hookrightarrow e' : \sigma' \times \tau'}{\Gamma \vdash \underline{\pi}_2\ e : \tau \hookrightarrow \pi_2\ e' : \tau'}$$

Note that even though pairs are dynamic, they can very well have static components. For example,

$$\begin{array}{l} \vdash \underline{\mathbf{let}}\ p = (2, \underline{4})\ \underline{\mathbf{in}}\ (\mathbf{lift}\ (\underline{\pi}_1\ p + 1), \underline{\pi}_2\ p) : \underline{\mathbf{int} \times \mathbf{int}} \\ \hookrightarrow \mathbf{let}\ p' = (\bullet, 4)\ \mathbf{in}\ (3, \pi_2\ p') : \mathbf{int} \times \mathbf{int} \end{array}$$

The residual type of p in this example is $2 \times \mathbf{int}$, which provides the information necessary to perform the static addition. After void erasure, the result becomes

$$\mathbf{let}\ p' = 4\ \mathbf{in}\ (3, p')$$

3.4 Tagged Sum Types

We introduce sum types in the form of tagged, n-ary sums, which we write as

$$C_1\ \tau_1 \mid C_2\ \tau_2 \mid \ldots \mid C_n\ \tau_n$$

We call the C_i the *constructors* of the type; the order in which we write them is insignificant. We will distinguish constructors lexically using an initial capital letter. Sometimes we shall write such a sum type as $\Sigma_{i=1}^n C_i\ \tau_i$.

Booleans are of course just a special case of a sum type, namely

$$\mathit{True}\ \mathbf{void} \mid \mathit{False}\ \mathbf{void}$$

We shall write *True* $\bullet$ and *False* $\bullet$ as **true** and **false** as usual, and use the ordinary **if-then-else** syntax as syntactic sugar for a **case** on this type.

We distinguish static and dynamic sum types. Dynamic sum types are identified by underlining (all of) the constructors.

[2] Binding a completely static variable with a dynamic <u>**let**</u> is not particularly useful, since the residual **let** will in any case be removed by void erasure. But binding a *partially static* variable with a dynamic <u>**let**</u> definitely is useful.

Static Sum Types Intuitively, if a value is of a static sum type, then we know at specialisation time which summand it lies in. Static constructor applications are specialised as follows:

$$\text{(SCON)} \quad \frac{\Gamma \vdash e : \tau \hookrightarrow e' : \tau'}{\Gamma \vdash C\ e : C\ \tau \mid \phi \hookrightarrow e' : C\ \tau'}$$

where ϕ varies over sum types only. Notice that the constructor is preserved in the residual type, and so it need not appear in the residual expression. The residual type $C\ \tau'$ therefore has the same elements as τ', but carries additional static information.

This information is then exploited in the rule for static **case** expressions:

$$\text{(SCASE)} \quad \frac{\begin{array}{c}\Gamma \vdash e : \Sigma_{i=1}^{n} C_i\ \sigma_i \hookrightarrow e' : C_k\ \sigma'_k \\ \Gamma, x_k : \sigma_k \hookrightarrow e' : \sigma'_k \vdash e_k : \tau \hookrightarrow e'_k : \tau'\end{array}}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ [C_i\ x_i \to e_i]_{i=1}^{n} : \tau \hookrightarrow e'_k : \tau'}$$

Here the residual type of the inspected expression tells us which constructor we have, so we can simply choose the right branch directly: the **case** expression leaves no trace in the residual program.

For example, using a tagged sum of integers and booleans

$$\mathit{Num}\ \underline{\mathbf{int}} \mid \mathit{Bool}\ \underline{\mathbf{bool}}$$

we could construct the expression

$$\begin{array}{l}\mathbf{case}\ \mathit{Num}\ (\underline{2+2})\ \mathbf{of}\ \ \mathit{Num}\ n \to \mathit{Num}\ (n \underline{\times} n) \\ \qquad\qquad\qquad\qquad\ \ \mathit{Bool}\ b\ \ \to \mathit{Bool}\ b\end{array}$$

Applying (SCON) to $\mathit{Num}\ (\underline{2+2})$ we obtain the specialisation $2+2 : \mathit{Num}\ \mathbf{int}$, and now applying (SCASE) we can choose the *Num* branch and derive the specialisation

$$(2+2) \times (2+2) : \mathit{Num}\ \mathbf{int}$$

as the result.

The example reveals that since the static **case** rule involves unfolding, code duplication may occur. But we can always avoid it by introducing a dynamic <u>**let**</u>:

$$\begin{array}{ll}\vdash & \underline{\mathbf{let}}\ m = \mathit{Num}\ (\underline{2+2}) \\ & \underline{\mathbf{in}}\ \mathbf{case}\ m\ \mathbf{of}\ \ \mathit{Num}\ n \to \mathit{Num}\ (n \underline{\times} n) \\ & \qquad\qquad\qquad\ \ \mathit{Bool}\ b\ \ \to \mathit{Bool}\ b \\ : & \mathit{Num}\ \underline{\mathbf{int}} \mid \mathit{Bool}\ \underline{\mathbf{bool}} \\ & \\ \hookrightarrow & \mathbf{let}\ m' = 2+2\ \mathbf{in}\ m' \times m' \\ : & \mathit{Num}\ \mathbf{int}\end{array}$$

Dynamic Sum Types In contrast, when a value is an element of a dynamic sum type we do *not* know which summand it lies in at specialisation time. The rule for constructor application 'forgets' this information, and inserts a constructor into the residual code:

$$\textit{(DCON)}\quad \frac{\Gamma \vdash e : \tau \hookrightarrow e' : \tau'}{\Gamma \vdash \underline{C}\; e : \underline{C}\; \tau \mid \phi \hookrightarrow C\; e' : C\; \tau' \mid \phi'}$$

The residual type is a sum of the residual types of the summands. Notice that here we concern ourselves only with the C summand; no relationship is imposed between ϕ and ϕ'. In practice however they must have the same constructors, and for each constructor the summand type in ϕ' must be a residual type obtainable from the corresponding summand in ϕ. This condition is enforced by many (DCON) rules, taken together.

The information in the residual type is then used to specialise the branches of dynamic $\underline{\textbf{case}}$ expressions, via the following rule.

$$\textit{(DCASE)}\quad \frac{\begin{array}{c}\Gamma \vdash e : \Sigma_{i=1}^{n} \underline{C_i}\; \sigma_i \hookrightarrow e' : \Sigma_{i=1}^{n} C_i\; \sigma_i' \\ [\Gamma, x_i : \sigma_i \hookrightarrow x_i' : \sigma_i' \vdash e_i : \tau \hookrightarrow e_i' : \tau']_{i=1}^{n}\end{array}}{\begin{array}{l}\Gamma \vdash \underline{\textbf{case}}\; e\; \underline{\textbf{of}}\; [\underline{C_i}\; x_i \to e_i]_{i=1}^{n} : \tau \hookrightarrow \\ \qquad \textbf{case}\; e'\; \textbf{of}\; [C_i\; x_i' \to e_i']_{i=1}^{n} : \tau'\end{array}}\quad x_i' \text{ fresh}, \mathrm{i} \in 1 \ldots \mathrm{n}$$

As an example of a dynamic sum type with static components, consider

$$\underline{\mathit{Num}}\; \textbf{int} \mid \underline{\mathit{Bool}}\; \textbf{bool}$$

with typical elements $\underline{\mathit{Num}}\; 42$ and $\underline{\mathit{Bool}}\; \textbf{true}$. These can for example be specialised as follows:

$$\begin{array}{l}\vdash \underline{\mathit{Num}}\; 42 : \underline{\mathit{Num}}\; \textbf{int} \mid \underline{\mathit{Bool}}\; \textbf{bool} \hookrightarrow \mathit{Num}\; \bullet : \mathit{Num}\; 42 \mid \mathit{Bool}\; \textbf{true} \\ \vdash \underline{\mathit{Bool}}\; \textbf{true} : \underline{\mathit{Num}}\; \textbf{int} \mid \underline{\mathit{Bool}}\; \textbf{bool} \hookrightarrow \mathit{Bool}\; \bullet : \mathit{Num}\; 42 \mid \mathit{Bool}\; \textbf{true}\end{array}$$

Both can be given the same residual type, which we can interpret as follows: we do not know at specialisation time if we have a *Num* or a *Bool*, but *if* we have a *Num*, then its static component is 42, and if we have a *Bool*, then its static component is **true**. When we specialise a corresponding $\underline{\textbf{case}}$ we can make use of this information, for example

$$\begin{array}{lll} b : \underline{\textbf{bool}} \hookrightarrow b' : \textbf{bool} \vdash & & \underline{\textbf{let}}\; x = \underline{\textbf{if}}\; b\; \underline{\textbf{then}}\; \underline{\mathit{Num}}\; 42\; \underline{\textbf{else}}\; \underline{\mathit{Bool}}\; \textbf{true} \\ & & \underline{\textbf{in}}\; \underline{\textbf{case}}\; x\; \underline{\textbf{of}}\; \underline{\mathit{Num}}\; n \to \textbf{lift}\; (n+1) \\ & & \qquad\qquad\qquad \underline{\mathit{Bool}}\; c\; \to \textbf{if}\; c\; \textbf{then}\; \underline{1}\; \textbf{else}\; \underline{0} \\ & : & \underline{\textbf{int}} \\ & & \\ & \hookrightarrow & \textbf{let}\; x' = \textbf{if}\; b'\; \textbf{then}\; \mathit{Num}\; \bullet\; \textbf{else}\; \mathit{Bool}\; \bullet \\ & & \textbf{in}\;\; \textbf{case}\; x'\; \textbf{of}\; \mathit{Num}\; n' \to 43 \\ & & \qquad\qquad\qquad \mathit{Bool}\; c' \;\to 1 \\ & : & \textbf{int}\end{array}$$

Here x has the residual type we discussed, and the $\underline{\textbf{case}}$ branches are specialised accordingly. Consequently the actual static components need not be preserved in the residual program.

Notice that a dynamic $\underline{\textbf{case}}$ expression may have a static or partially static result. For example (remembering that **if-then-else** is syntactic sugar for **case**),

$$\underline{\textbf{if}}\ b\ \underline{\textbf{then}}\ \mathit{Num}\ \underline{3}\ \underline{\textbf{else}}\ \mathit{Num}\ \underline{4} : \mathit{Num}\ \underline{\textbf{int}} \mid \mathit{Bool}\ \underline{\textbf{bool}}$$

is well-typed and specialises to

$$\textbf{if}\ b\ \textbf{then}\ 3\ \textbf{else}\ 4 : \mathit{Num}\ \textbf{int}$$

As expected, the static tags are eliminated from the program, and the static information about the branches is still available in the residual type of the whole. Such an effect is sometimes achieved in traditional partial evaluators using indirect methods such as continuations; here, it is a natural consequence of using type inference.

But this begs the question: what if the static information in the branches fails to match? For example, the expression

$$\underline{\textbf{if}}\ b\ \underline{\textbf{then}}\ \mathit{Num}\ \underline{3}\ \underline{\textbf{else}}\ \mathit{Bool}\ \underline{\textbf{true}} : \mathit{Num}\ \underline{\textbf{int}} \mid \mathit{Bool}\ \underline{\textbf{bool}}$$

is also well-typed, but the two branches have *different* static constructors, and therefore different residual types. No specialisation is possible here: we would not be able to assign the residual program a type. Our specialiser therefore rejects this expression.

Notice that this is not a binding-time error: the input expression is well typed, no static value appears where a dynamic one is expected. The error can only be discovered during specialisation, when two residual types fail to match.

3.5 Function Types

We shall also distinguish static and dynamic function types. Dynamic $\underline{\lambda}$-expressions create λ-expressions in the residual programs, whereas static functions are unfolded.

Dynamic Functions The rules for dynamic $\underline{\lambda}$-expressions and applications are quite straightforward extensions of the usual typing rules.

$$\textit{(DLAM)}\quad \frac{\Gamma, x : \sigma \hookrightarrow x' : \sigma' \vdash e : \tau \hookrightarrow e' : \tau'}{\Gamma \vdash \underline{\lambda} x.e : \sigma \underline{\rightarrow} \tau \hookrightarrow \lambda x'.e' : \sigma' \rightarrow \tau'}\ \ x' \text{ fresh}$$

$$\textit{(DAPP)}\quad \frac{\Gamma \vdash e_1 : \sigma \underline{\rightarrow} \tau \hookrightarrow e_1' : \sigma' \rightarrow \tau' \qquad \Gamma \vdash e_2 : \sigma \hookrightarrow e_2' : \sigma'}{\Gamma \vdash e_1 \mathbin{\underline{@}} e_2 : \tau \hookrightarrow e_1'\ e_2' : \tau'}$$

We have already seen examples in section 2 showing that dynamic functions may well have static arguments and results, in contrast to unfolding based partial evaluators.

We also introduce a dynamic fixpoint operator, which generates a residual **fix** in the specialised program.

$$\textit{(DFIX)}\quad \frac{\Gamma \vdash e : \tau \underline{\rightarrow} \tau \hookrightarrow e' : \tau' \rightarrow \tau'}{\Gamma \vdash \underline{\textbf{fix}}\ e : \tau \hookrightarrow \textbf{fix}\ e' : \tau'}$$

Static Functions Static λ-expressions do not generate a function in the residual program; instead they are unfolded by the partial evaluator. Static functions are therefore represented by 'closures' at specialisation time, containing the function body, bound variable, and the context. But such a closure is a mixture of static and dynamic information: although the names, types, and residual types of the variables in the context are all static, their *values* may be dynamic. We therefore transform a static function into a residual *tuple* of the values of the variables in the context, and assign it a residual type containing the remaining parts of the function closure. The specialisation rule for static λ-expressions is therefore

$$\textit{(SLAM)} \quad \begin{array}{l}[z_i : \tau_i \hookrightarrow e_i : \tau_i']_{i=1}^n \ \vdash \\ \qquad \lambda x.e : \sigma \to \tau \hookrightarrow (e_1, \ldots, e_n) : \mathbf{clos} < [z_i : \tau_i \hookrightarrow \tau_i']_{i=1}^n, x, e >\end{array}$$

For example, consider the program

$$\begin{array}{l}\mathbf{let}\ m = 42 \\ \mathbf{in}\ \ \mathbf{let}\ n = \underline{35} \\ \qquad \mathbf{in}\ \ \underline{\mathbf{let}}\ f = \lambda x.x \underline{+} n\ \underline{\mathbf{in}}\ (f\ \underline{1}, f\ \underline{2})\end{array}$$

Here the static λ-expression is in the scope of two variables, and so its residual expression is a pair:

$$\begin{array}{l}m : \mathbf{int} \hookrightarrow \bullet : 42, n : \underline{\mathbf{int}} \hookrightarrow 35 : \mathbf{int} \ \vdash \\ \quad \lambda x.x \underline{+} n : \underline{\mathbf{int}} \to \underline{\mathbf{int}} \ \hookrightarrow \\ \qquad (\bullet, 35) : \mathbf{clos} < [m : \mathbf{int} \hookrightarrow 42, n : \underline{\mathbf{int}} \hookrightarrow \mathbf{int}], x, x \underline{+} n >\end{array}$$

This tuple of values is passed around in the residual program instead of the function value, and when a static function is applied the residual type tells us how to interpret it. We extract the function's body from the closure and specialise it, binding the variables in its context to appropriate components of the tuple:

$$\textit{(SAPP)} \quad \frac{\begin{array}{c}\Gamma \vdash e_1 : \sigma \to \tau \hookrightarrow e_1' : \mathbf{clos} < [z_i : \tau_i \hookrightarrow \tau_i']_{i=1}^n, x, e > \\ \Gamma \vdash e_2 : \sigma \hookrightarrow e_2' : \sigma' \\ [z_i : \tau_i \hookrightarrow \pi_i\ e_1' : \tau_i']_{i=1}^n, x : \sigma \hookrightarrow e_2' : \sigma' \vdash e : \tau \hookrightarrow e' : \tau'\end{array}}{\Gamma \vdash e_1\ e_2 : \tau \hookrightarrow e' : \tau'}$$

In our example, when f's body is specialised at the calls, the free variable n is replaced by the second component of the pair. The result of specialising the complete program is therefore

$$\begin{array}{l}\mathbf{let}\ f' = (\bullet, 35) \\ \mathbf{in}\ \ (1 + \pi_2\ f', 2 + \pi_2\ f')\end{array}$$

Of course void erasure is applicable, and eliminates the projections altogether in this case:

$$\mathbf{let}\ f' = 35\ \mathbf{in}\ (1 + f', 2 + f')$$

Our treatment of static functions is very similar to Similix's [Bon91], which replaces them by their dynamic free variables. We don't distinguish dynamic from static *variables*, and the rule above takes all the variables in the context, not just the free ones, but otherwise the idea is the same. We can achieve an effect closer to Similix by adding a weakening rule to drop unused variables from the context before (SLAM) is applied:

$$(WEAK) \quad \frac{\Gamma_1, \Gamma_2 \vdash e : \tau \hookrightarrow e' : \tau'}{\Gamma_1, x : \sigma \hookrightarrow u' : \sigma', \Gamma_2 \vdash e : \tau \hookrightarrow e' : \tau'} \; x \notin FV(e)$$

In our implementation we use weakening to remove all but the free variables before building a static closure. So in the example, the value of f is actually specialised as

$$\begin{array}{l} m : \mathbf{int} \hookrightarrow \bullet : 42, n : \underline{\mathbf{int}} \hookrightarrow 35 : \mathbf{int} \;\vdash \\ \quad \lambda x.x \,\underline{+}\, n : \underline{\mathbf{int}} \rightarrow \underline{\mathbf{int}} \;\hookrightarrow \\ \qquad 35 : \mathbf{clos} < [n : \underline{\mathbf{int}} \hookrightarrow \mathbf{int}], x, x \,\underline{+}\, n > \end{array}$$

in which the unused variable m is dropped.

Static Recursion We express static recursion, which is unfolded at specialisation time, via a fixpoint operator on static functions. But we cannot allow an unrestricted **fix**, because expressions such as **fix** $(\lambda x.\underline{Cons}\ (\underline{1}, x))$ would lead to infinite unfolding at specialisation time. Instead we restrict ourselves to recursive *functions*, and we delay unfolding until a recursive call is actually made. Of course, this is the same solution adopted in strict programming languages.

We cannot conveniently represent recursive functions using the kind of closure we have discussed so far. The difficulty is that the body of a recursive function has the function name itself as a free variable. In the residual program it would have to be represented by a tuple with one component representing the function itself — in other words, equal to the whole tuple. To avoid the need for cyclic structures in the residual program, we introduce a new kind of static closure:

$$\mathbf{rec} < f, \Gamma', x, e >$$

Here f is the recursive function name, and is implicitly bound to the entire closure. The corresponding value in the residual program is just a tuple of the *other* free variables.

Recursive closures are created by the static **fix** rule. The argument of **fix** must be a static function from a static function type $\sigma \rightarrow \tau$ to itself. The (SFIX) rule unfolds that function, thereby specialising the body of the recursive definition, *without* a binding for its parameter, the recursively defined name f. Of course in principle f is in scope in the body, *but* if the body cannot be specialised without reference to it, then the recursion is ill-defined — it would lead to infinite unfolding. In practice we expect the body to be another static λ-expression, which we can specialise just by constructing a static closure. This closure will probably refer to f in its body, but will not contain a binding for f because of the way it is constructed. We convert it into a **rec** closure, thus binding f, and obtain the final result. Here is the specialisation rule, followed by an example:

$$(SFIX) \quad \frac{\begin{array}{l} \Gamma \vdash e_1 : (\sigma \rightarrow \tau) \rightarrow (\sigma \rightarrow \tau) \;\hookrightarrow \\ \qquad e_1' : \mathbf{clos} < [z_i : \alpha_i \hookrightarrow \alpha_i']_{i=1}^{m}, f, e_2 > \\ [z_i : \alpha_i \hookrightarrow \pi_i\ e_1' : \alpha_i']_{i=1}^{m} \;\vdash \\ \quad e_2 : \sigma \rightarrow \tau \hookrightarrow e_2' : \mathbf{clos} < [w_i : \beta_i \hookrightarrow \beta_i']_{i=1}^{n}, x, e_3 > \end{array}}{\Gamma \vdash \mathbf{fix}\ e_1 : \sigma \rightarrow \tau \hookrightarrow e_2' : \mathbf{rec} < f, [w_i : \beta_i \hookrightarrow \beta_i']_{i=1}^{n}, x, e_3 >}$$

As an example, consider the program

$$\begin{array}{l} \mathbf{let}\ n = \underline{35} \\ \mathbf{in}\ \ \underline{\mathbf{let}}\ f = \mathbf{fix}\ (\lambda g.\lambda x.\mathbf{if}\ x = 0\ \mathbf{then}\ n\ \mathbf{else}\ g\ (x-1) \underline{+} \underline{1}) \\ \qquad \underline{\mathbf{in}}\ f\ 2 \end{array}$$

Here the argument of **fix** specialises as follows:

$$\begin{array}{l} n : \underline{\mathbf{int}} \hookrightarrow 35 : \mathbf{int}\ \vdash \\ \quad \lambda g.\lambda x.\mathbf{if}\ x = 0\ \mathbf{then}\ n\ \mathbf{else}\ g\ (x-1) \underline{+} \underline{1} : (\mathbf{int} \rightarrow \underline{\mathbf{int}}) \rightarrow (\mathbf{int} \rightarrow \underline{\mathbf{int}}) \ \hookrightarrow \\ \qquad 35 : \mathbf{clos} < [n : \underline{\mathbf{int}} \hookrightarrow \mathbf{int}], g, \lambda x.\mathbf{if}\ x = 0\ \mathbf{then}\ n\ \mathbf{else}\ g\ (x-1) \underline{+} \underline{1} > \end{array}$$

When we unfold this closure without a binding for g, we obtain

$$\begin{array}{l} n : \underline{\mathbf{int}} \hookrightarrow 35 : \mathbf{int}\ \vdash \\ \quad \lambda x.\mathbf{if}\ x = 0\ \mathbf{then}\ n\ \mathbf{else}\ g\ (x-1) \underline{+} \underline{1} : \mathbf{int} \rightarrow \underline{\mathbf{int}} \ \hookrightarrow \\ \qquad 35 : \mathbf{clos} < [n : \underline{\mathbf{int}} \hookrightarrow \mathbf{int}], x, \mathbf{if}\ x = 0\ \mathbf{then}\ n\ \mathbf{else}\ g\ (x-1) \underline{+} \underline{1} > \end{array}$$

in which g is a free variable. Applying (SFIX), the final result of specialising the **fix** is

$$35 : \mathbf{rec} < g, [n : \underline{\mathbf{int}} \hookrightarrow \mathbf{int}], x, \mathbf{if}\ x = 0\ \mathbf{then}\ n\ \mathbf{else}\ g\ (x-1) \underline{+} \underline{1} >$$

which is the recursive function we want.

We also need a rule to apply recursive functions: it is just the same as (SAPP), except that it additionally binds the recursive function name to the very function being applied.

$$\textit{(RSAPP)} \quad \frac{\begin{array}{c} \Gamma \vdash e_1 : \sigma \rightarrow \tau \hookrightarrow e_1' : \mathbf{rec} < f, [z_i : \tau_i \hookrightarrow \tau_i']_{i=1}^{n}, x, e > \\ \Gamma \vdash e_2 : \sigma \hookrightarrow e_2' : \sigma' \\ \left\{ \begin{array}{l} [z_i : \tau_i \hookrightarrow \pi_i\ e_1' : \tau_i']_{i=1}^{n}, \\ f : \sigma \rightarrow \tau \hookrightarrow e_1' : \mathbf{rec} < f, [z_i : \tau_i \hookrightarrow \tau_i']_{i=1}^{n}, x, e >, \\ x : \sigma \hookrightarrow e_2' : \sigma' \end{array} \right\} \vdash e : \tau \hookrightarrow e' : \tau' \end{array}}{\Gamma \vdash e_1\ e_2 : \tau \hookrightarrow e' : \tau'}$$

Using this rule we can complete the specialisation of our example. The residual program is:

$$\mathbf{let}\ f = 35\ \mathbf{in}\ (f+1)+1 : \mathbf{int}$$

The static recursion has been unfolded twice, while the function name is bound to a (one-)tuple of its free variables, which is used to interpret the reference to n in the unfolded code.

3.6 Recursive Types

We use no special notation for recursive types, instead we simply allow both source and residual types to be regular trees. For example, the type of static lists is

$$\mathit{List}\ \tau \equiv \mathit{Nil}\ \mathbf{void} \mid \mathit{Cons}\ (\tau \underline{\times} \mathit{List}\ \tau)$$

Since the list constructors are static, they do not appear in residual programs. For example,

$$\begin{aligned}&\vdash \mathit{Cons}\ (\underline{1}, \mathit{Cons}\ (\underline{2}, \mathit{Nil}\ \bullet)) : \mathit{List}\ \underline{\mathbf{int}}\ \hookrightarrow\\ &\quad (1,(2,\bullet)) : \mathit{Cons}\ (\mathbf{int} \times \mathit{Cons}\ (\mathbf{int} \times \mathit{Nil}\ \mathbf{void}))\end{aligned}$$

The residual type tells us the length of the list, while the residual term is just a (nested) tuple of the list elements.

For an example of a recursive residual type, consider the term $\underline{\mathbf{fix}}\ (\underline{\lambda} l.\mathit{Cons}\ (\underline{1}, l))$, which evaluates to an infinite list of ones. It specialises as follows:

$$\vdash \underline{\mathbf{fix}}\ (\underline{\lambda} l.\mathit{Cons}\ (\underline{1}, l)) : \mathit{List}\ \underline{\mathbf{int}} \hookrightarrow \mathbf{fix}\ (\lambda l'.(1, l')) : \tau$$

where

$$\tau \equiv \mathit{Cons}\ (\mathbf{int} \times \tau)$$

3.7 Polyvariance

So far, we have actually described a form of *monovariant* specialisation — that is, one in which static variables can take only *one* static value. For example, the expression

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

can be specialised to

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

But the similar expression

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

cannot be specialised at all, because f cannot be assigned both residual types $3 \rightarrow \mathbf{int}$ and $4 \rightarrow \mathbf{int}$. All useful partial evaluators use polyvariant specialisation, and in this case would create two versions of f, one specialised to each static argument.

Fortunately, our partial evaluator can be extended by adding a new type whose specialisation rules are polyvariant. It is introduced by the operator **poly**:

$$\text{(POLY)}\quad \frac{[\Gamma \vdash e : \tau \hookrightarrow e_i : \tau_i']_{i=1}^{n}}{\Gamma \vdash \mathbf{poly}\ e : \mathbf{poly}\ \tau \hookrightarrow (e_1, \ldots, e_n) : (\tau_1', \ldots, \tau_n')}$$

An expression enclosed by **poly** can be specialised many times, with different residual types in each case. In our example, we will enclose the value of f by **poly** so that it can be specialised to two different arguments.

When we do so, however, we change the type of f to $\mathbf{poly}\ (\mathbf{int} \rightarrow \underline{\mathbf{int}})$, which is not a function and so cannot be applied directly. We need another operator, **spec**, to extract a τ from a $\mathbf{poly}\ \tau$. The corresponding residual expression selects an element from the tuple of specialisations that **poly** produces.

$$\text{(SPEC)}\quad \frac{\Gamma \vdash e : \mathbf{poly}\ \tau \hookrightarrow e' : (\tau_1', \ldots, \tau_n')}{\Gamma \vdash \mathbf{spec}\ e : \tau \hookrightarrow \pi_k\ e' : \tau_k'}$$

In our example, we apply **spec** at the two uses of f, giving

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

Now f can be given the residual type $(3 \to \mathbf{int}, 4 \to \mathbf{int})$, resulting in the specialisation

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

or after void erasure,

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

3.8 Polyvariant Sums

The polyvariance just introduced creates an n-ary *product* in the residual program; it is natural to ask whether there is a corresponding form which creates an n-ary *sum*. Indeed there is, and moreover, its rules can be derived just by 'reversing the arrows' in the rules above! Of course this would be clearer if we used a categorical notation rather than the λ-calculus, but even so, the rules below are simply the duals of those in the previous section.

We introduce a polyvariant sum type $\mathbf{sum}\ \tau$, with constructor $\underline{In}$, which we take apart using a $\underline{\mathbf{case}}$ expression. Expressions of type $\mathbf{sum}\ \tau$ yield residual expressions with an n-ary sum type of the form $\Sigma_{i=1}^{n} In_i\ \tau'_i$. Applications of $\underline{In}$ are specialised to applications of an appropriate In_i. Compare the rule below to (SPEC):

$$\textit{(SPECCON)}\quad \frac{\Gamma \vdash e : \tau \hookrightarrow e' : \tau'_k}{\Gamma \vdash \underline{In}\ e : \mathbf{sum}\ \tau \hookrightarrow In_k\ e' : \Sigma_{i=1}^{n} In_i\ \tau'_i}$$

$\underline{In}$ can be applied to expressions with *different* residual types to produce expressions with the *same* one, thus providing another way to pass different static arguments to the same function. The example of the previous section can instead be rewritten as

$$\begin{array}{l} \underline{\mathbf{let}}\ f = \underline{\lambda} x.\underline{\mathbf{case}}\ x\ \underline{\mathbf{of}}\ \underline{In}\ y \to \mathbf{lift}\ (y+1) \\ \underline{\mathbf{in}}\ (f\ \underline{@}(\underline{In}\ 3), f\ \underline{@}(\underline{In}\ 4)) \end{array}$$

Here f can be given the residual type $(In_1\ 3\ \mid In_2\ 4) \to \mathbf{int}$ and the two actual parameters specialised as

$$\begin{array}{l} \vdash \underline{In}\ 3 : \mathbf{sum}\ \mathbf{int} \hookrightarrow In_1\ \bullet : In_1\ 3\ \mid In_2\ 4 \\ \vdash \underline{In}\ 4 : \mathbf{sum}\ \mathbf{int} \hookrightarrow In_2\ \bullet : In_1\ 3\ \mid In_2\ 4 \end{array}$$

When a $\underline{\mathbf{case}}$ expression on $\mathbf{sum}\ \tau$ is specialised, then the residual type of the inspected expression tells us which specialised branches the residual **case** should contain. The following rule is dual to (POLY):

$$\textit{(POLYCASE)}\quad \frac{\begin{array}{c}\Gamma \vdash e_1 : \mathbf{sum}\ \sigma \hookrightarrow e'_1 : \Sigma_{i=1}^{n} In_i\ \sigma'_i \\ [\Gamma, x : \sigma \hookrightarrow x' : \sigma'_i \vdash e_2 : \tau \hookrightarrow e'_{2,i} : \tau']_{i=1}^{n}\end{array}}{\begin{array}{l}\Gamma \vdash \underline{\mathbf{case}}\ e_1\ \underline{\mathbf{of}}\ \underline{In}\ x \to e_2 : \tau\ \hookrightarrow \\ \qquad\qquad \mathbf{case}\ e'_1\ \mathbf{of}\ [In_i\ x' \to e'_{2,i}]_{i=1}^{n} : \tau'\end{array}}\quad x' \text{ fresh}$$

Applied to the example above, we obtain

$$\begin{array}{l} \textbf{let } f' = \lambda x'.\textbf{case } x' \textbf{ of } \; In_1 \; y' \to 4 \\ \qquad\qquad\qquad\qquad\qquad\quad\; In_2 \; y' \to 5 \\ \textbf{in } (f \; (In_1 \; \bullet), f \; (In_2 \; \bullet)) \end{array}$$

This completes our specification of the specialiser itself. In the following sections we shall describe the phases that precede and follow the actual specialisation.

4 Type-Checking the Source Language

One way to think of the specialisation rules we have given is that they type-check the source and residual expressions simultaneously. For the source language, well-typing guarantees among other things that dynamic values are not treated as static, and vice versa. But we do not wish to delay type-checking source programs until during specialisation: partly because specialisation may not visit every part of the source program, and therefore might fail to discover some type-errors, and partly because specialisation can be time-consuming and we want to report type errors fast.

We therefore type-check source terms before specialisation. Type correctness guarantees that when we do apply the specialisation rules, any failures to match will involve residual types, not source types.

The typing rules are straightforward, and will not be presented in detail. Most of them can be obtained from the corresponding specialisation rules by erasing the residual terms and types. Occasionally this does not suffice: for example, the rule for static **case** expressions

$$(SCASE) \qquad \frac{\Gamma \vdash e : \Sigma_{i=1}^{n} C_i \; \sigma_i \hookrightarrow e' : C_k \; \sigma'_k \qquad \Gamma, x_k : \sigma_k \hookrightarrow e' : \sigma'_k \vdash e_k : \tau \hookrightarrow e'_k : \tau'}{\Gamma \vdash \textbf{case } e \textbf{ of } [C_i \; x_i \to e_i]_{i=1}^{n} : \tau \hookrightarrow e'_k : \tau'}$$

specialises only *one* branch of the **case**, corresponding to the residual type of the inspected expression. The typing rule, in contrast, checks *all* branches, and so verifies that every invocation of the specialisation rule will match.

$$(T\text{-}SCASE) \qquad \frac{\Gamma \vdash e : \Sigma_{i=1}^{n} C_i \; \sigma_i \qquad [\Gamma, x_i : \sigma_i \vdash e_i : \tau]_{i=1}^{n}}{\Gamma \vdash \textbf{case } e \textbf{ of } [C_i \; x_i \to e_i]_{i=1}^{n} : \tau}$$

Similarly the specialisation rule for **poly** specialises the enclosed expression many times, while the typing rule only types it once. The specialisation rules for static λ-expressions arrange to unfold them at the point of application, whereas the typing rule is just the usual rule for λ, which verifies that *all* unfoldings will be well-typed. There are no surprises here.

Indeed, the only surprising thing is that the typing rules are so unsurprising. Other authors using two-level languages restrict the ways in which static and dynamic type formers are mixed — for example Nielson and Nielson do so by distinguishing *compile-time* from *run-time* types[NN92]. In contrast we can allow completely free type formation. Type-based binding-time analyses (such as Henglein

[Hen91]) must usually handle 'dependency constraints' which force the result type of a conditional expression to be dynamic if the condition is. In contrast we can allow a dynamic conditional to be of any type whatsoever — even a static integer! Our 'binding-time checker' is simply an ordinary type-checker.

Note that we use a binding-time *checker*, not a binding-time *analyser*: all binding-times are explicitly given by the programmer. We will discuss prospects for binding-time analysis in a later section.

5 Void Erasure

We have seen that residual programs contain many trivial values without computational content. Most of these can be removed by a post-processing phase we call *void erasure*.

We define a predicate τ triv, which holds of 'trivial' types such as 3, **void**, and $3 \to 4$. Whenever τ triv holds, then $\tau \simeq \mathbf{void}$. We define τ triv as follows:

$$\begin{gathered}\mathbf{void}\ \mathrm{triv}\\ m\ \mathrm{triv}[\mathrm{m\ an\ integer}]\\ \sigma \to \tau\ \mathrm{triv} \mathrel{\hat{=}} \tau\ \mathrm{triv}\\ \sigma \times \tau\ \mathrm{triv} \mathrel{\hat{=}} \sigma\ \mathrm{triv} \wedge \tau\ \mathrm{triv}\\ \Sigma_{i=1}^{n} \tau_i\ \mathrm{triv} \mathrel{\hat{=}} \mathrm{n} = 0\end{gathered}$$

We treat n-ary products as nested pairs, and the empty product as **void** for the purposes of void erasure.

The triv predicate is the *largest* predicate with these properties, which means that recursive types such as

$$\mathit{Str}\ \tau \equiv \tau \times \mathit{Str}\ \tau$$

can be trivial (in this case provided τ is).

Void erasure consists of removing trivial components from pairs, trivial parameters from functions, and variables bound to trivial values. We are in fact exploiting the isomorphisms

$$\begin{gathered}\mathbf{void} \times \tau \simeq \tau\\ \tau \times \mathbf{void} \simeq \tau\\ \mathbf{void} \to \tau \simeq \tau\end{gathered}$$

by transforming terms of the more complex types on the left into the corresponding terms of the simpler types on the right. The fact that these are isomorphisms guarantees that void erasure does not change the behaviour of residual programs.

We transform terms depending on their types, which we write as superscripts here. Let ν by a trivial type. The following transformations are applied wherever they are applicable:

$$\begin{gathered}e^{\nu} \longrightarrow \bullet\\ \mathbf{let}\ x^{\nu} = e^{\nu}\ \mathbf{in}\ e'^{\tau} \longrightarrow e'\end{gathered}$$

It is safe to remove the binding for x, because after applying the first rule no references to x can remain.

$$\lambda x^{\nu}.e \longrightarrow e$$
$$e^{\nu \to \tau}\ e'^{\nu} \longrightarrow e$$

We erase trivial parameters, both formal and actual.

$$(e^{\nu}, e'^{\tau}) \longrightarrow e' \qquad \pi_1\ e^{\tau \times \nu} \longrightarrow e$$
$$(e^{\tau}, e'^{\nu}) \longrightarrow e \qquad \pi_2\ e^{\nu \times \tau} \longrightarrow e$$

We erase trivial components from pairs, and we must obviously remove the corresponding selector applications also.

These transformations cannot be applied during specialisation, because they depend on knowing residual types. Just as a type-checker may discover the type of an expression some time after it has been visited, so our specialiser may discover a residual type some time after constructing the corresponding residual expression. Hence void erasure must be done in a later pass.

The specialiser also has a tendency to produce terms of the form $\pi_i\ (e_1, \ldots, e_n)$, especially from programs involving static functions. We simplify these terms also during void erasure, although we view this as a hack. The correct solution to this problem would be to introduce a form of static product, whose projections are removed by the specialiser. Doing so, however, is not completely straightforward (see section 8.1).

After void erasure some occurrences of $\bullet$ may remain, but only in contexts where the rules above cannot remove them, such as the argument of a constructor. Our insistance on unary constructors means that such occurrences are really necessary: sometimes void types really are useful!

We note in passing that our assumption of a lazy language is helpful here. Indeed, the void erasure transformation is not safe for a strict language, since it transforms λ-expressions which always terminate into other expressions which may not. A more careful transformation, and more complicated justification, would be needed to apply void erasure in a strict context.

6 Implementing the Specialisation Rules

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. Since the source expressions are checked for type correctness in advance, our specialiser does not infer source types, only residual expressions and types.

Just as type inference often produces types containing uninstantiated type variables, so our specialiser often produces residual types containing uninstantiated variables. For example, if we specialise $\underline{Num}\ \underline{3}$ then the result is

$$\vdash \underline{Num}\ \underline{3} : \underline{Num}\ \underline{\mathbf{int}} \mid \phi \hookrightarrow Num\ 3 : Num\ \mathbf{int} \mid \phi'$$

where ϕ' is an uninstantiated residual type variable. Our specialiser maintains a state containing the next free variable and the variable instantiations known so far, just as a type checker does. Residual types are derived using unification, as usual. In order to support cyclic residual types 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 are specialised using the rule

$$\textit{(SCASE)}\quad \frac{\begin{array}{c}\Gamma \vdash e : \Sigma_{i=1}^{n} C_i\ \sigma_i \hookrightarrow e' : C_k\ \sigma'_k \\ \Gamma, x_k : \sigma_k \hookrightarrow e' : \sigma'_k \vdash e_k : \tau \hookrightarrow e'_k : \tau'\end{array}}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ [C_i\ x_i \to e_i]_{i=1}^{n} : \tau \hookrightarrow e'_k : \tau'}$$

which specialises only *one* of the branches, depending on which summand the residual type of the expression inspected lies in. The implementation specialises this expression first, and if its residual type is $C_k \sigma'_k$ it chooses the kth branch. But unfortunately it is quite possible that the residual type is only an uninstantiated variable! In this case we cannot choose the branch to specialise until later, when the variable becomes instantiated. We have implemented a 'demon' mechanism, which allows a computation to be suspended until a particular variable is instantiated. This mechanism is used in this case.

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.

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

$$\begin{array}{ll}\lambda x.\mathbf{case}\ x\ \mathbf{of} & \mathit{Num}\ a \to \underline{3} \\ & \mathit{Bool}\ b\ \ \to \underline{4}\end{array}$$

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.

Demons are used in many other cases where rule application depends on residual types that might not be known when the specialiser reaches the term in question. For example, to wait for the arguments of primitive static operations, to wait for the value of a **lift**, or to wait for the static closure at a static function application.

The most problematic rules to implement are those for polyvariant products and sums. The problems are the same in both cases: in rules (POLY) and (POLYCASE) we do not know which variants to create, and in rules (SPEC) and (SPECCON) we do not know which variant to choose.

We have solved these problems by introducing *indexed sets*, of the form

$$\{(i, \tau_i), (j, \tau_j), \ldots\}$$

to collect the residual types of the variants. Residual n-ary sum and product types are represented as **sum** Φ and **poly** Φ respectively, where Φ is such a set with indexes from 1 to n. The rule we use for (SPEC) becomes

$$\textit{(SPEC)} \quad \frac{\Gamma \vdash e : \mathbf{poly}\ \tau \hookrightarrow e' : \mathbf{poly}\ (\{(k, \tau')\} \cup \Phi)}{\Gamma \vdash \mathbf{spec}\ e : \tau \hookrightarrow \pi_k\ e' : \tau'}$$

When we apply this rule we leave k uninstantiated, thus deferring the choice of index until later. Φ, the rest of the set, is also uninstantiated at this stage.

When the same polyvariant value is specialised several times, we have to unify sets of the form $\{(i, \tau_i)\} \cup \Phi_1$ and $\{(j, \tau_j)\} \cup \Phi_2$. This is achieved by instantiating Φ_1 and Φ_2: the result is $\{(i, \tau_i), (j, \tau_j)\} \cup \Phi$, where Φ is a new set variable. It is always possible to unify two sets of this form.

The rule for (POLY) becomes

$$\textit{(POLY)} \quad \frac{[\Gamma \vdash e : \tau \hookrightarrow e_i : \tau'_i]_{(i,\tau'_i) \in \Phi}}{\Gamma \vdash \mathbf{poly}\ e : \mathbf{poly}\ \tau \hookrightarrow (e_1, \ldots, e_n) : \mathbf{poly}\ \Phi}$$

In the implementation, we create a demon which waits for Φ to be instantiated, and then takes each element as it becomes available and constructs an appropriate specialisation of e.

One can compare the behaviour of (SPEC) to adding a new entry to the pending list in a traditional partial evaluator, and (POLY) to scanning the pending list and constructing appropriate specialisations. Of course, we also want to avoid constructing multiple specialisations with the same residual type. But we cannot simply compare two elements τ'_i and τ'_j of a set, because they might both be partially unknown. Instead, we *unify* each new set element with the previously seen ones: if unification succeeds, then we proceed on the assumption that those two elements are equal and only one specialisation is required. But if unification fails, or if we later encounter an error, then we backtrack to this point and try the next element, or construct a new specialisation instead.

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 residual type 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 adding an element to the set that was almost completely uninstantiated; all pairs of set elements 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 arguments, and in the case of a **spec** expression the function types added to the set had different, known arguments so that the wrong alternatives could usually be eliminated at once. 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!

For the same reason, we delay unifying set elements with one another until *after* one of them has been used to generate a specialisation in the (POLY) rule. Generating such a specialisation tends to run demons which are waiting to instantiate the residual type further, and thus helps avoid initially successful unifications which will only lead to failure when specialisations are actually generated.

Finally, once all other specialisation is complete, we assign indexes 1 to n to the elements of each set. This cannot be done earlier since it is always possible that any two sets Φ_1 and Φ_2 will be unified; if one has begun assigning indexes already in each set, then the assignments may be inconsistent.

The implementation is written in Haskell, and uses a *monad*[Wad92] to provide the mechanisms discussed here, namely a state, demons, and backtracking.

7 Optimal Specialisation of Interpreters

The acid test for our partial evaluator is whether it can remove an entire layer of interpretation, including eliminating type tags. To demonstrate that it can, we specialise a very simple interpreter for the simply typed λ-calculus with constants.

7.1 The Universal Type

In the interpreter, we represent values by elements of the following universal type:

$$\mathit{Univ} \equiv \mathit{Num}\ \underline{\mathbf{int}} \mid \mathit{Fun}\ (\mathit{Univ} \mathrel{\underline{\to}} \mathit{Univ})$$

We expect integer and function values to appear in residual programs, hence these types are dynamic, but we expect the *type tags* themselves to disappear. Hence we use static constructors.

An example of an expression of this type is

$$\mathit{Fun}\ \left(\underline{\lambda} x.\mathbf{case}\ x\ \mathbf{of}\ \begin{array}{ll} \mathit{Num}\ y \to & \mathit{Num}\ (y \underline{+} \underline{1}) \\ \mathit{Fun}\ z \to & \ldots \end{array}\right) : \mathit{Univ}$$

It can be specialised to

$$\lambda x'.x' + 1 : \mathit{Fun}\ (\mathit{Num}\ \mathbf{int} \to \mathit{Num}\ \mathbf{int})$$

as follows:

- The outer *Fun* constructor becomes part of the residual type.
- The dynamic $\underline{\lambda}$ creates a residual λ-expression.
- Assume we know from the context that x has residual type *Num* **int**. Then the static **case** chooses the first branch.
- The *Num* constructor becomes part of the residual type.
- The dynamic addition creates a residual addition.

The residual code here contains no tagging and untagging operations. Moreover, the type of the residual expression is essentially **int** $\to$ **int**, since the *Fun* and *Num* constructors do not change the elements of a residual type. Type specialisation has been achieved. And indeed, it is clear that any type at all built from **int** and function arrow can be obtained as a residual type from *Univ*.

7.2 Expressions

The expressions the interpreter evaluates are of course purely static: we represent them by the type

$$E \equiv \mathit{Cn}\ \mathbf{int} \mid \mathit{Vr}\ \mathbf{int} \mid \mathit{Lm}\ (\mathbf{int} \times E) \mid \mathit{Ap}\ (E \times E) \mid \mathit{Fx}\ E$$

Because our partial evaluator does not handle strings, we represents variable names by integers — but these are just used as distinct names, not de Bruijn numbers.

7.3 The Environment

The environment in the interpreter is represented as a function mapping variable names to values. It may of course be applied to many different (static) variable names. We have a choice between using a *static* function, which is unfolded at each use, and a *polyvariant dynamic* function, which can be specialised to look up many different names. That is, we can choose between the types

$$\begin{gathered} \mathit{Env} \equiv \mathbf{int} \rightarrow \mathit{Univ} \\ \mathit{Env} \equiv \mathbf{poly}\ (\mathbf{int} \underline{\rightarrow}\ \mathit{Univ}) \end{gathered}$$

Which choice we make affects the representation of residual environments in the specialised program. With a static function type, residual environments will be tuples of values of the variables in scope, and with a polyvariant type they will be tuples of functions specialised to looking up the variables used. But in the latter case, since the arguments of these functions are completely static, they will be removed by void erasure. So in both cases a residual environment is just a tuple of variable values. The difference is that in the former case all variables in scope appear, and in the latter case only those variables which are actually looked up.

However, in the interpreter below residual environments only appear at variable lookups, where selection from the tuple is simplified to a single element in any case, so the difference is not significant.

7.4 An Optimal Interpreter

The interpreter itself appears below.

$$\begin{array}{l}
(\mathbf{fix}\ (\lambda \mathit{eval}.\lambda \mathit{env}.\lambda e. \\
\quad \mathbf{case}\ e\ \mathbf{of} \\
\qquad \begin{array}{ll}
\mathit{Cn}\ n & \rightarrow \mathit{Num}\ (\mathbf{lift}\ n), \\
\mathit{Vr}\ i & \rightarrow \mathit{env}\ i, \\
\mathit{Lm}\ (i, e) & \rightarrow \mathit{Fun}\ (\underline{\lambda} v.\mathbf{let}\ \mathit{env}' = \lambda j.\mathbf{if}\ i = j\ \mathbf{then}\ v\ \mathbf{else}\ \mathit{env}\ j \\
 & \qquad\qquad\quad \mathbf{in}\ \ \mathit{eval}\ \mathit{env}'\ e), \\
\mathit{Ap}\ (e_1, e_2) & \rightarrow \mathbf{case}\ \mathit{eval}\ \mathit{env}\ e_1\ \mathbf{of}\ \mathit{Fun} f \rightarrow f\ \underline{@}(\mathit{eval}\ \mathit{env}\ e_2), \\
\mathit{Fx}\ e & \rightarrow \mathbf{case}\ \mathit{eval}\ \mathit{env}\ e\ \mathbf{of}\ \mathit{Fun}\ f \rightarrow \underline{\mathbf{fix}}\ f))
\end{array} \\
(\lambda i.\,\mathit{Wrong}\ \bullet)
\end{array}$$

It is a partial application of the *eval* function to the initial environment, that is, a function from closed expressions to their values. We have taken the liberty of pattern matching on pairs, rather than using explicit selectors: this is purely syntactic sugar.

We have chosen to unfold applications of *eval*, and of the environment. The case dispatch on the expression to evaluate is static and disappears of course, as do the type tags *Num* and *Fun*. Indeed, the only dynamic operations in this interpreter are **lift**, used to create constants, the dynamic $\underline{\lambda}$ used to create function values, the dynamic application used to interpret *Ap*, and the dynamic $\underline{\mathbf{fix}}$ used to interpret *Fx*.

It is clear that this interpreter can be specialised optimally. Just to take one or two examples, if we specialise it to

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

we obtain

$$(\lambda v.v)\ 3 : Num\ \mathbf{int}$$

If we specialise it to

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

we obtain

$$(\lambda v.v\ 3)\ (\lambda v.v) : Num\ \mathbf{int}$$

If we specialise it to

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

which binds variable 1 to the identity function and then applies it both to itself and a number, then we obtain a specialisation time error:

$$\text{Cannot unify } Num\ \alpha \text{ with } Fun\ \beta.$$

Of course this is quite right: the program is not type correct because variable 1 is applied to both a number and a function.

7.5 A Firstifying Interpreter

It is interesting to consider variations on the little interpreter above. For example, suppose we choose to make function values static also, so that we take the universal type to be

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

The effect will be to unfold user function applications also, and to replace function values in the residual program by tuples of their free variables. In other words, the result is a first-order program, and the specialiser is doing firstification.

However, this choice for the universal type is very restrictive. It requires that all functions that can reach the same point be closures of the same λ-expression, otherwise the residual types will not match. We can relax this requirement using a polyvariant sum:

$$Univ = Num\ \underline{\mathbf{int}} \mid Fun\ (\mathbf{sum}\ (Univ \to Univ))$$

Now functions are replaced by elements of a sum type, where the constructor identifies the function, and the component is a tuple of free variables. In other words, an efficient representation of a closure. With this choice of universal type, residual type inference discovers which functions might appear at each point — that is, it does a form of type-based closure analysis.

Rather than unfold user functions at each call, which would lead to code duplication, we introduce a dynamic function *app* which we use to apply function values:

$$app \equiv \underline{\lambda} f.\underline{\lambda} x.\mathbf{case}\ f\ \mathbf{of}\ \mathit{Fun}\ g \rightarrow \underline{\mathbf{case}}\ g\ \underline{\mathbf{of}}\ \underline{\mathit{In}}\ h \rightarrow h\ x$$

In residual programs, the specialisations of *app* will contain a case for applying each possible closure.

We have given user functions a polyvariant sum type, so that one specialisation of *app* can be used to apply many different functions. But we still have to make *app* itself polyvariant, because its *second* parameter x may take different residual types. The firstifying interpreter is therefore:

$$\begin{array}{l}
\underline{\mathbf{let}}\ app = \mathbf{poly}\underline{\lambda} f.\underline{\lambda} x.\mathbf{case}\ f\ \mathbf{of}\ \mathit{Fun}\ g \rightarrow \underline{\mathbf{case}}\ g\ \underline{\mathbf{of}}\ \underline{\mathit{In}}\ h \rightarrow h\ x \\
\underline{\mathbf{in}}\ (\mathbf{fix}\ (\lambda eval.\lambda env.\lambda e. \\
\qquad \mathbf{case}\ e\ \mathbf{of} \\
\qquad\quad \begin{array}{ll}
\mathit{Cn}\ n & \rightarrow \mathit{Num}\ (\mathbf{lift}\ n), \\
\mathit{Vr}\ i & \rightarrow env\ i, \\
\mathit{Lm}\ (i, e) & \rightarrow \mathit{Fun}\ (\underline{\mathit{In}}\ (\lambda v.\mathbf{let}\ env' = \lambda j.\mathbf{if}\ i = j\ \mathbf{then}\ v\ \mathbf{else}\ env\ j \\
& \qquad\qquad\qquad\quad \mathbf{in}\ \ eval\ env'\ e)), \\
\mathit{Ap}\ (e_1, e_2) & \rightarrow \mathbf{spec}\ app\ \underline{@}(eval\ env\ e_1)\ \underline{@}(eval\ env\ e_2), \\
\mathit{Fx}\ e & \rightarrow \underline{\mathbf{fix}}\ (\mathbf{spec}\ app\ \underline{@}(eval\ env\ e))
\end{array} \\
\quad (\lambda i.\ \mathit{Wrong}\ \bullet)
\end{array}$$

We interpret λ-expressions using a static λ, injected into a polyvariant sum type, and we interpret application (and **fix**) using the *app* function we discussed.

When this interpreter is specialised to, for example,

$$\begin{array}{l}
\mathit{Ap}\ (\ \mathit{Lm}\ (1, \mathit{Ap}\ (\ \mathit{Lm}\ (2, \mathit{Ap}\ (\ \mathit{Vr}\ 1,\ \mathit{Vr}\ 2)), \\
\qquad\qquad\qquad\qquad\quad \mathit{Ap}\ (\ \mathit{Vr}\ 1,\ \mathit{Cn}\ 3))), \\
\qquad \mathit{Lm}\ (3,\ \mathit{Vr}\ 3))
\end{array}$$

the residual program is

$$\begin{array}{l}
\mathbf{let}\ app = (\ \lambda f.\lambda x.\mathbf{case}\ f\ \mathbf{of}\ \mathit{In}_1\ h \rightarrow \pi_2\ h\ (\mathit{In}_2\ (x, h))\ (\pi_2\ h\ x\ 3), \\
\qquad\qquad\quad \lambda f.\lambda x.\mathbf{case}\ f\ \mathbf{of}\ \mathit{In}_1\ h \rightarrow x, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathit{In}_2\ h \rightarrow \pi_2\ (\pi_2\ h)\ (\pi_1\ h)\ x) \\
\mathbf{in}\ \pi_1\ app\ (\mathit{In}_1\ app)\ (\mathit{In}_1\ app)
\end{array}$$

In this example there are two types of function, and hence two specialisations of *app*. The first can apply $\mathit{Lm}\ (1, \ldots)$, and the second can apply both $\mathit{Lm}\ (2, \ldots)$ and $\mathit{Lm}\ (3,\ \mathit{Vr}\ 3)$, represented as $\mathit{In}_2\ h$ and $\mathit{In}_1\ h$ respectively.

Notice that every closure contains a reference to *app*. This is not surprising since *app* is indeed a free variable of *eval*, and therefore of each function denotation, but since it is a global variable we might want to avoid building it into every closure. We can do so by passing *app* as an explicit parameter to *eval*, and to each function denotation. We need to make the definition of *app* recursive:

$$
\begin{array}{l}
\underline{\textbf{let}}\ app = \underline{\textbf{fix}}\ \underline{\lambda} app.\textbf{poly}\underline{\lambda} f.\underline{\lambda} x. \\
\qquad\qquad\quad \textbf{case}\ f\ \textbf{of}\ Fun\ g \rightarrow \underline{\textbf{case}}\ g\ \underline{\textbf{of}}\ \underline{In}\ h \rightarrow h\ app\ x \\
\underline{\textbf{in}}\ (\textbf{fix}\ (\lambda eval.\lambda app.\lambda env.\lambda e. \\
\qquad \textbf{case}\ e\ \textbf{of} \\
\qquad\quad \begin{array}{ll}
Cn\ n & \rightarrow Num\ (\textbf{lift}\ n), \\
Vr\ i & \rightarrow env\ i, \\
Lm\ (i, e) & \rightarrow Fun\ (\underline{In}\ (\lambda app'.\lambda v. \\
 & \qquad\qquad \textbf{let}\ env' = \lambda j.\textbf{if}\ i = j\ \textbf{then}\ v\ \textbf{else}\ env\ j \\
 & \qquad\qquad \textbf{in}\ \ eval\ app'\ env'\ e)), \\
Ap\ (e_1, e_2) & \rightarrow \textbf{spec}\ app\ \underline{@}(eval\ app\ env\ e_1)\ \underline{@}(eval\ app\ env\ e_2), \\
Fx\ e & \rightarrow \underline{\textbf{fix}}\ (\textbf{spec}\ app\ \underline{@}(eval\ app\ env\ e))
\end{array} \\
\quad app \\
\quad (\lambda i.\,Wrong\ \bullet)
\end{array}
$$

Notice that we interpret λ-expressions as functions of app', so that *app* is no longer a free variable of the static closures we build. When we specialise this interpreter to the previous example, we obtain

$$
\begin{array}{l}
\textbf{let}\ app = \textbf{fix}\lambda app.(\lambda f.\lambda x.\textbf{case}\ f\ \textbf{of}\ In_1\ h \rightarrow \pi_2\ app\ (In_2\ x)\ (\pi_2\ app\ x\ 3), \\
\qquad\qquad\qquad\qquad \lambda f.\lambda x.\textbf{case}\ f\ \textbf{of}\ In_1\ h \rightarrow x, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad In_2\ h \rightarrow \pi_2\ app\ h\ x)) \\
\textbf{in}\ \pi_1\ app\ (In_1\ \bullet)\ (In_1\ \bullet)
\end{array}
$$

in which the closures now contain only the variables we would expect.

8 Discussion

The partial evaluator we have presented represents a first attempt to base specialisation on type inference. There are certainly a number of areas where improvements or extensions are desirable. In this section we shall discuss further work, and also the pros and cons of inference-based specialisation as a whole.

8.1 Static Products and Arity Raising

Just as we have static sum types whose constructors and **case**s are removed by the specialiser, so it would be natural to add a static product type whose constructor and selectors are removed. We shall write static pairs as $<x, y>$ in this section.

Static products have a natural connection to *arity raising* [Rom90]; when a static pair is passed to a function, the pair structure can be removed by currying the function:

$$f\ \underline{@} <x, y> \hookrightarrow f\ x\ y$$

Similarly constructors applied to static pairs can be curried, although this does oblige us to introduce n-ary constructors in residual programs. **let** bindings of static pairs can be transformed into n-ary **let**s in residual programs:

$$
\underline{\textbf{let}}\ x = <y, z>\ \underline{\textbf{in}}\ e \hookrightarrow \begin{array}{l}\textbf{let}\ \ x_1 = y \\ \qquad\ x_2 = z \\ \textbf{in}\ e\end{array}
$$

But in other contexts the static pair structure can be more awkward to eliminate. For example, when the result of a dynamic conditional is a static pair, the best we can do is to move the pair structure outward where it can hopefully be removed by one of the transformations above.

$$\underline{\textbf{if}}\ b\ \underline{\textbf{then}} <x, y> \underline{\textbf{else}} <u, v> \hookrightarrow <\textbf{if}\ b\ \textbf{then}\ x\ \textbf{else}\ u, \textbf{if}\ b\ \textbf{then}\ y\ \textbf{else}\ v>$$

This transformation leads to code duplication, as do several others involving static pairs.

Adding static pairs requires the addition of a large number of transformation rules to move them outwards and eliminate them, and it is not clear that they fit nicely into the specialisation rule framework we have presented. We have therefore left them for future work.

8.2 Polymorphism

Both the source and residual languages of our specialiser are simply typed. Allowing polymorphism in source programs seems reasonably easy, with the caveat that for a polymorphic function to be useful it must be applicable at different source types, and therefore at different residual types, which requires that it also be polyvariant. But generating polymorphic residual programs seems much more difficult.

The standard Milner polymorphic type inference algorithm relies on generalising type variables which remain uninstantiated after an expression's type is inferred. The corresponding strategy in our specialiser would be to generalise residual type variables which remain uninstantiated after an expression's specialisation is complete. But because of the demon mechanism, we do not know when specialisation of an expression *is* complete! The standard strategy is therefore inapplicable here.

Our inability to produce polymorphic residual programs is a serious deficiency. Clearly, optimal specialisation of polymorphic programs cannot be achieved until this restriction is lifted.

8.3 Controlling Polyvariance

Our **poly** construct gives us only crude control over polyvariance. For example, in an interpreter for a simply typed language we might make the *eval* function 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! There is a dual problem for polyvariant sums.

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

8.4 Non-termination

A related problem with our treatment of polyvariance is that it can cause the specialiser to loop on erroneous inputs, rather than report an error. For example, consider the program

$$\underline{\mathbf{let}}\ f = \underline{\mathbf{fix}}\ (\underline{\lambda} f.\mathbf{poly}\ \underline{\lambda} n.(\mathbf{lift}\ n, \mathbf{spec}\ f\ n))\ \underline{\mathbf{in}}\ \mathbf{spec}\ f\ 2$$

which specialises to

$$\mathbf{let}\ f = \mathbf{fix}\ (\lambda f.(2, f))\ \mathbf{in}\ f$$

Here both occurrences of **spec** f refer to the same specialisation; our implementation succeeds in unifying their residual types, and in producing the result shown. *But*, if a specialisation error is encountered later, we assume that it may be caused by this unification! We therefore backtrack and try constructing two specialisations of f instead. But of course, in this case this doesn't help. Our specialiser falls into a loop, constructing more and more identical specialisations of f, rather than reporting the error.

To avoid this we would need to use some kind of dependency-directed backtracking, where we only undo a unification if it actually contributed to the detected error. It is not clear how to do this either.

8.5 Dead Code and Subtyping

Our specialiser sometimes generates residual programs containing dead code. For example, consider the program

$$\begin{array}{l}
\underline{\mathbf{let}}\ f = \underline{\lambda} x.\underline{\mathbf{case}}\ x\ \underline{\mathbf{of}}\ \underline{In}\ y \rightarrow \mathbf{lift}\ (y+1) \\
\underline{\mathbf{in}}\ \underline{\mathbf{let}}\ g = \underline{\lambda} x.\underline{\mathbf{case}}\ x\ \underline{\mathbf{of}}\ \underline{In}\ y \rightarrow \mathbf{lift}\ (y+2) \\
\quad \underline{\mathbf{in}}\ \underline{\mathbf{let}}\ x = \underline{In}\ 3 \\
\qquad \underline{\mathbf{in}}\ f\ x \underline{+} g\ x \underline{+} g\ (\underline{In}\ 4)
\end{array}$$

which is specialised to

$$\begin{array}{l}
\mathbf{let}\ f = \lambda x.\mathbf{case}\ x\ \mathbf{of}\ \begin{array}[t]{l} In_1\ y \rightarrow 4 \\ In_2\ y \rightarrow 5 \end{array} \\
\mathbf{in}\ \ \mathbf{let}\ g = \lambda x.\mathbf{case}\ x\ \mathbf{of}\ \begin{array}[t]{l} In_1\ y \rightarrow 5 \\ In_2\ y \rightarrow 6 \end{array} \\
\quad \mathbf{in}\ \ \mathbf{let}\ x = In_1\ \bullet \\
\qquad \mathbf{in}\ \ f\ x + g\ x + g\ (In_2\ \bullet)
\end{array}$$

Here f contains a case for $In_2\ y$, even though it is never applied to such an argument. This case branch is therefore dead code.

The reason the dead code is generated is that g is applied to both x and $\underline{In}\ 4$, and therefore its residual type must be $(In_1\ 3 \mid In_2\ 4) \rightarrow \mathbf{int}$, which forces the residual type of x to be $In_1\ 3 \mid In_2\ 4$. Now since f is also applied to x, it must have the same residual type as g, and so must contain cases for the same arguments.

Dead code is a problem in partial evaluation, because it is possible that generating the dead branch might lead to a specialisation time error, and so to failure of the whole specialisation. In this particular case it may be possible to avoid it by introducing *subtyping* on residual types. One would then give x the residual type $In_1\ 3$, which would be a subtype of $In_1\ 3 \mid In_2\ 4$, thus permitting x to be passed to g without forcing it to have the larger type, and in turn forcing dead code to be generated in f.

We have explained this problem in terms of polyvariant sums, but a dual example can be constructed using polyvariant products, and a similar approach using subtyping offers hope of a solution here too.

8.6 Self-application

Self-application of our partial evaluator is still some way off. To achieve it, we will need to redesign our monad carefully in order to separate the static and meta-static information held as the values of residual type variables. When the partial evaluator is specialised, residual programs will of course use a similar monad, with all the mechanism of demons, uninstantiated variables, and backtracking. Generated compilers will pass all data around via unification: reasonable enough for types, but a little curious for the program being compiled! It is interesting to wonder whether we could replace unification by a simpler, more functional mechanism in some cases.

8.7 Correctness of Partial Evaluation

We have specified a program specialiser, but we have not proved that there is any relation at all between the *semantics* of source programs, and the semantics of their specialisations! We will need to define an interpretation for specialisation judgements, and prove the inference rules sound. We leave this proof for future work.

8.8 Specialisation-Time Errors

Our specialiser is unusual in that well-typed source programs may give rise to specialisation time errors (when residual types fail to match). One referee considered that this revealed a weakness in the two-level type system:

> "It is the very definition of well-annotated two-levelness that specialisation cannot go wrong. The fact that your specialiser can reject some of its input just shows an inadequacy between your specialiser and the binding-time annotations of its source programs."

We argue that, on the contrary, the possibility of specialisation time errors is inherent in optimal specialisation of typed languages.

When a partial evaluator is used for compiling by specialising an interpreter, we can distinguish three interesting times:

- *run-time*, when the compiled code (specialised interpreter) is run,
- *compile-time*, when the interpreter is specialised to a particular program,
- *compiler-generation time*, when the interpreter is pre-processed to prepare it for specialisation.

Binding-time analysis, and any other checking of the interpreter in the absence of a particular program to specialise it to, is done at compiler-generation time. Is it reasonable to apply any criterion at compiler-generation time, which guarantees that no errors can occur at compile-time?

We claim the answer is no: it is an inherent property of compilers for typed programming languages that they refuse to compile ill-typed programs. Indeed, it is precisely *because* they reject ill-typed programs that they are able to generate code for well-typed ones that does not manipulate type tags. No analysis at compiler generation time can guarantee that all programs to be compiled will be well-typed; type errors cannot be detected earlier than compile-time.

Analogously, an optimal specialiser for typed programs must reject some inputs at specialisation time. No binding-time analysis can guarantee that specialisation time errors will not occur.

8.9 Binding-time Analysis

Our specialiser processes completely annotated programs; the programmer decides whether each value will be static or dynamic. Certainly we find a binding-time *checker* very useful, which ensures that the programmer's annotations are consistent. But most other partial evaluators use a binding-time *analyser* to decide automatically which expressions are dynamic, given a division of the program's inputs into dynamic and static. What are the prospects for coupling a binding-time analysis to our partial evaluator?

Unfortunately, we believe they are not very good. Our partial evaluator is of a different nature to other offline ones, and is less well suited to an automatic choice of binding times. Every partial evaluator admits a certain amount of freedom in this choice — there may be different ways to annotate the same program consistently. Thus one may be able to choose whether a particular expression should be static or dynamic. With a conventional partial evaluator, it is always better to choose static[3]. Binding-time analysers therefore choose the 'most static' consistent annotation. But in our case, although making types more static certainly improves the results of specialisation, it can also lead to specialisation-time errors.

For example, in the λ-calculus interpreter in section 7.4 we chose to make the universal type *Univ* a *static* sum type. As a consequence type tags are eliminated from residual programs, but at the cost of restricting the λ-terms that can be compiled by specialising this interpreter to be well-typed. If we had instead chosen to use a *dynamic* sum type, then we could have compiled all λ-terms, but type tags and checks would have remained in the residual code. In other words, the choice of binding-time here determines whether we compile a statically or dynamically typed λ-calculus. It is hard to see that such a choice can be made automatically.

On the other hand, a less ambitious form of binding-time analysis may be possible. For example, if λ-expressions are annotated static or dynamic, then perhaps applications need not be, since one can infer from the type of the function which

[3] Unless this leads to large or unbounded static variation — see Jones' article in this volume.

kind of application is intended. In general, if the programmer provides enough annotations that there is only one consistent way to complete them, then that completion can safely be constructed automatically.

8.10 Is Unification Desirable in a Partial Evaluator?

One may ask whether the mechanisms in our partial evaluator are overkill: maybe optimal specialisation of typed programs could be achieved in a simpler way? We think not.

Suppose we generate a compiler for a typed language by specialising a partial evaluator to a particular interpreter. If the generated compiler is not to insert type checks in the code that it generates, then it must perform type inference. To do so, it must propagate type information via unification. But type information in the generated compiler is of course just static information in the partial evaluator. It follows that the partial evaluator must propagate static information at least as effectively as unification does. We consider it very natural that a partial evaluator that can be specialised to obtain a compiler that performs type inference, should itself be based on the mechanisms of type inference.

9 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 representation of dynamic values.

Continuation-based specialisation (invented by Consel and Danvy [CD91], and further developed by Bondorf [Bon92]) also enables dynamic conditionals to yield static results, by moving the context (represented as a continuation) into the branches. A dynamic conditional in a context C

$$C[\underline{\mathbf{if}}\ b\ \underline{\mathbf{then}}\ e_1\ \underline{\mathbf{else}}\ e_2]$$

is specialised instead as

$$\underline{\mathbf{if}}\ b\ \underline{\mathbf{then}}\ C[e_1]\ \underline{\mathbf{else}}\ C[e_2]$$

Obviously, if e_1 and e_2 are static, then they can be used statically in C. But in contrast to our case, it is not an error for e_1 and e_2 to have different static values — the context is simply specialised twice.

However, a continuation-based specialiser cannot be used to remove type tags. The trouble is that the context C can only be specialised to e_1 or e_2 if it is itself static: if the context simply applies a dynamic continuation then no specialisation is possible, and e_1 and e_2 are themselves forced to be dynamic. Because unboundedly many continuations may arise during an execution of most programs, making all continuations static would lead to non-termination at specialisation time. So some continuations must be dynamic, and so must their arguments therefore — including any type tags. This is enough to force type tags to be dynamic everywhere.

Mogensen has proposed a form of type specialisation which he calls *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

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

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

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

instead of the original. All **case** expressions matching on *Cons* must 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.

Mogensen's constructor specialisation was the inspiration for our polyvariant sums (section 3.8), which provide essentially the same mechanism.

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 residual types. 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. Perhaps this is why it requires approximations to ensure 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[4]. This is not surprising since the partial evaluator never rejects inputs as ours does; ill-typed programs can be compiled by the generated compilers. 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 are 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, we need only compare residual function *types*, not functions themselves, and so polyvariant specialisation is unproblematic.

10 Conclusions

We have presented a new paradigm for partial evaluation, inspired by type inference. Our partial evaluator is specified in terms of *specialisation rules* that prescribe how a source expression and type should be transformed into a residual expression and type. Static information is represented in the residual types, in contrast to traditional partial evaluators which effectively represent it in the transformed expression. Thus we break the link between staticness and unfolding: a variable can be static even if the specialiser does not substitute a value for it; static information is propagated by unification, not by substitution. As a result we obtain better static information flow, and consequently stronger specialisation, than traditional partial evaluators.

We can describe our specialiser in a very modular way: each specialisation feature is tied to a particular type in the source language, with associated introduction and elimination rules. Each feature is specified independently, and features can be

[4] This specialiser is however optimal for a language with *only* constructor types — because then all values have some kind of tag, which need not be eliminated. See Mogensen's paper in this volume.

combined in arbitrary ways. In contrast to earlier two-level languages, we can allow free formation of types. This has led to a number of new results.

Our partial evaluator performs type specialisation: this fits naturally into the framework, since there is no reason to expect source and residual types to be the same. Ours is the first specialiser which can obtain an *arbitrary* type in the residual program by specialising one universal type in the source program.

We can allow dynamic functions to take partially static arguments and return partially static results; this is the key to optimal specialisation of typed programs, where the static information represents type information. Ours is the first optimal specialiser for a typed, higher-order language.

Ours is also the first specialiser to support both higher-order functions and constructor specialisation. This combination is powerful: we can obtain closure analysis and firstification as a simple application.

We conclude that inference-based specialisation is a very promising new approach. But our prototype specialiser suffers from a number of drawbacks, and we are already aware of several desirable extensions. There is much more work to be done.

Finally, although we have described our approach in terms of the λ-calculus, the basic idea is quite general. We see no reason why similar results should not be obtainable for other typed languages, including imperative ones.

Acknowledgements

This is an appropriate place to thank Neil Jones: not only has his pioneering work inspired myself and many others over the years, but one of his talks on a very different approach to type specialisation was the direct stimulus to the work described here. Neil, Olivier Danvy, and the anonymous referees provided exceptionally good comments on the draft of this paper, and Dirk Dussart and Peter Thiemann offered useful insights that have improved the final version. I am grateful to them all.

References

[Bon91] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17(1-3):3–34, December 1991. Selected papers of ESOP '90, the 3rd European Symposium on Programming.

[Bon92] Anders Bondorf. Improving binding times without explicit cps-conversion. In *1992 ACM Conference on Lisp and Functional Programming. San Francisco, California*, pages 1–10, June 1992.

[CD91] Charles Consel and Olivier Danvy. For a Better Support of Static Data Flow. In John Hughes, editor, *Functional Programming and Computer Architecture*, LNCS, pages 496–519. Springer-Verlag, 1991.

[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.

[Hen91] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *FPCA*, pages 448–472. 5th ACM Conference, Cambridge, MA, USA, Springer-Verlag, August 1991. Lecture Notes in Computer Science, Vol. 523.

[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.

[NN92] F. Nielson and H. R. Nielson. *Two-Level Functional Languages.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[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.

[Wad92] P. Wadler. The essence of functional programming. In *Proceedings 1992 Symposium on principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, 1992.

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

This article was processed using the LaTeX macro package with LLNCS style