A Type Specialisation Tutorial

John Hughes

Department of Computing Science, Chalmers University of Technology, S-41296 Göteborg, SWEDEN. `rjmh@cs.chalmers.se`

1 Introduction

The essence of partial evaluation is beautifully simple: we just take a program, together with values of some of its inputs; we perform the operations that depend only on known inputs, build a new program from the other operations, and finally obtain a residual program which solves the same problem as the original for a subclass of the cases. Work by Neil Jones and his group over the past decade and a half has demonstrated just how powerful this simple idea really is.

But unfortunately, partial evaluation suffers from a serious problem: it fits badly with types. Since most real programming languages are typed, this is a major drawback. The problem is that the basic partial evaluation mechanism, evaluation and simplification of expressions, does not take types into account. As a result, simple partial evaluators cannot specialise types; residual programs operate on just the same types as the original program. But in principle, we might expect to gain just as much by specialising the data structures used in a program, as by specialising its code. While various forms of type specialisation have been added to partial evaluators in the past, the techniques needed are complicated, and the results not as far reaching as one might wish.

Type specialisation [4, 3, 1, 5] is a new approach to program specialisation which integrates types into the basic specialiser mechanisms, so that specialisation of data-structures can be achieved just as simply as specialisation of code. In this chapter we give a tutorial introduction to the subject via examples in a simple functional language.

How to use this tutorial. Much of the material in this chapter is contained in exercises, which encourage you to use the type specialiser in ways which syntax-directed partial evaluators cannot emulate. The intention is that readers solve the exercises on a machine while reading this chapter, and in our examples we therefore use the concrete ASCII syntax which the implemented specialiser accepts. The specialiser is written in Haskell 1.4, and the source code is available on the World Wide Web; for greatest convenience, download and compile the specialiser to use on your own machine. Readers without access to a Haskell 1.4 compiler may instead solve the exercises by running the specialiser on the Chalmers University web server. The URL is

`http://www.cs.chalmers.se/~rjmh/TypeSpec2`

2 Type Specialisation Basics

A partial evaluator specialises an expression to produce a residual expression, for example `2 + 2` specialises to `4`. As in this example, if the original expression is static, then the residual expression will be a constant.

In contrast, the type specialiser specialises an expression *and type* to a residual expression *and residual type*. Thus both source and residual programs are typed. But while source types look much like the types in any other functional language, except that static and dynamic types are distinguished, the residual types carry very precise information. In this example, `2 +@ 2` (using the type specialiser's syntax) specialises to

```
Residual type: 4

Residual code:
void
```

Here the residual type, `4`, is a 'singleton type' which tells us the value of the expression precisely. And since the value is explicit in the residual type, it need not appear at all in the residual code, which is just a dummy expression `void`.

This example illustrates one of the basic principles of type specialisation: static information is always carried in *residual types*, and consequently it need never appear in *residual terms*. We will see later how this property enables us to achieve stronger specialisations than partial evaluation can.

The example also illustrates another property of type specialisation: source programs are explicitly annotated to distinguish static from dynamic operations. Static operations are indicated with an '`@`' symbol; in this case `+@` is the static addition operator. Unlike most partial evaluators, there is no binding-time analyser to insert such annotations automatically; we will discuss the reason for this later.

Let us consider a very simple example, to show how types can be used to propagate static information during specialisation. Consider the annotated expression

```
(\f. lift (f 3)) (\x. x +@ 1)
```

Here `\x.e` is our notation for a λ-expression, and juxtaposition is function application. In this example, the λs and the applications are dynamic, and so will not be reduced by the specialiser. The `lift` operator just modifies binding-times; it converts a static integer into a dynamic one.

Given this input, the type specialiser specialises `3` (which is static) to `void` with residual type `3`; the static information is carried by the type. Since `f` is applied to this expression, it must have residual type `3 -> a` for some type `a`. But since the λ-expression `\x...` is bound to `f`, it must have the same residual type `3 -> a`. It follows that `x` has residual type `3`. Given this information, `x +@ 1` can be specialised to `void` with residual type `4` (once again, the static information is carried in the type). So now we know that `\x...`, and consequently also `f`, has residual type `3 -> 4`. As a result `f 3` can be specialised to `f void` with residual type `4`, and then the `lift ...` can be specialised to `4`, with residual type `int`. The *type* of `lift`'s argument tells us what residual *code* we should generate for it. Since the result of `lift` is dynamic, the only static information expressed in its residual type is that it is an integer. The final result is

```
(\f. 4) (\x. void)
```

with residual type `int`.

Type specialisation often produces programs containing 'dummy' void expressions, as in this case; indeed, every static expression specialises to one. The implementation therefore includes a post-processor called the *arity raiser*, which deletes every expression and every variable binding with a void type. (Actually, as we will see, arity raising is a little more general than this). In this example, it deletes the binding of `f` and the corresponding actual parameter, producing the final result `4`.

The interesting thing about this example is that the λ-expression `\x. ...` can be specialised, and can produce a static result, even though it is a dynamic function passed as a parameter to another dynamic function. When static information is propagated via types, there is no need to unfold a function application in order to produce a static result. Consequently the type specialiser can propagate more static information than a partial evaluator can, and thus achieve stronger specialisation.

Indeed, this example *as annotated* could not be specialised by a partial evaluator. Binding-time analysis imposes constraints on the ways in which binding-times are used, for example, that dynamic functions must return dynamic results. That constraint is broken in this example, and so this annotation could never be produced by binding-time analysis. In contrast, the type specialiser allows any binding-time to be used anywhere (provided, of course, they are used consistently; this is enforced by assigning static and dynamic values different types in the source language). In particular, dynamic functions may very well have static or partly static results.

In this particular example, there is no reason to make the λ-expressions dynamic; by annotating them static instead, then the same final result could be achieved by partial evaluation. But in general, a specialiser cannot unfold *every* function call, and the ability of the type specialiser to return static information even from the calls which are not unfolded gives significant extra power.

3 Polyvariance

Let us consider a slight variation on the example above:

```
(\f. f 3 + f 4) (\x. lift (x +@ 1))
```

When we try to specialise this term, we obtain the slightly surprising error message

```
Error: Cannot unify 4 with 3
```

The problem is that the arguments of `f` in the two calls have *different* residual types, namely `3` and `4`, and `f` cannot be assigned both `3 -> ...` and `4 -> ...` as its type. Of course, our intention is to specialise `f` twice, but the type specialiser will not do so without further annotations.

Duplication by Unfolding. One possibility is to make the λs and applications static, so that they are unfolded. Our notation is to add `@` to both λs and applications:

```
(\@f. f@3 + f@4)@(\@x. lift (x +@ 1))
```

Since unfolding duplicates the body of `\@x...` at two places, it can be specialised twice with different residual types for the bound variable `x`. The result is

```
4 + 5
```

But this is hardly satisfactory in general: we would not wish to have to unfold *every* function which is applied to several different static arguments. Instead, we would like to specialise `f` *polyvariantly*, to produce several residual functions rather than just one.

Polyvariant Specialisation. Polyvariance is introduced via an explicit annotation: the expression `poly e` specialises to a *tuple* of specialisations of `e`, with a residual type which is a product of the residual types of the individual specialisations. In the source language, `poly e` has the type `poly t`, where `t` is the type of `e`. Consequently, a value created by `poly` cannot be used directly as a value of type `t`; to recover such a value we use the operator `spec`. When `e` is polyvariant, and so specialises to a tuple, then `spec e` specialises to an appropriate selection from the tuple. Using `poly` and `spec` we can annotate our example as

```
(\f. spec f 3 + spec f 4) (poly \x. lift (x +@ 1))
```

which specialises to

```
(\f. (fst f) void + (snd f) void) (\x.4, \x.5)
```

in which `f` has the residual type `(3 -> int, 4 -> int)`.

The tuples introduced by polyvariance quickly make residual programs unreadably messy; we therefore let the arity raiser eliminate not only void values, but also all tuples. In this case, the result of arity raising is

```
(\f_1.\f_2.f_1 + f_2) 4 5
```

where the void values bound to `x` have been removed, and the residual pair that `f` specialised to has been replaced by two separate parameters. The relationship to traditional arity raising in partial evaluation is clear.

Constructor Specialisation. There is actually a third way to specialise this example. Instead of generating two residual versions of `f` with different argument types, we may generate a single version whose argument type is a *sum* of the two types. We introduce such polyvariant sums using the special constructor `In`; just as `poly e` specialises to a residual expression with a product type, so `In e` specialises to an expression with a sum type, with one summand for each type of value which flows together. Values are extracted from sum types by pattern matching; a `case` over a polyvariant sum specialises to a `case` with one branch for each residual summand. This is just a version of Mogensen's constructor specialisation [6].

In our example, we tag the two different arguments to `f` with `In`, so that they specialise to elements of a common sum type, and we strip off the tag by pattern matching in the body of `f`:

```
(\f. f (In 3) + f (In 4))
  (\z. case z of In x: lift (x +@ 1) esac)
```

In the residual program, the two occurrences of `In` specialise to different constructors, and the `case` specialises to a `case` with two branches:

```
(\f.f (In1 void) + f (In0 void))
  (\z.case z of In0 x : 5, In1 x : 4 esac)
```

The type of `z` in this case is `In0 4 | In1 3`, which as usual carries all the static information: it records the two values passed to `f`, and their association with the specialised constructors.

Finally, the arity raiser deletes the void components, and the corresponding bound variable `x`, to produce

```
(\f.f In1 + f In0)
  (\z.case z of In0 : 5, In1 : 4 esac)
```

4 Syntactic Sugar

Now it is time to put these techniques into practice. As a first example, we shall employ them to specialise the ubiquitous `power` function. But first, we must introduce a little more syntax: the minimal language we have introduced so far cannot even express `power` palatably!

It is tedious to bind names with λs the whole time, so we introduce a form of `let` definitions. The example in the previous section can then be written more readably as

```
let f = \x. lift (x +@ 1) in
  f 3 + f 4
```

or, with the usual syntactic sugar for λ, as

```
let f x = lift (x +@ 1) in
  f 3 + f 4
```

This defines `f` as a dynamic, or non-unfoldable function. To make `f` unfoldable, we write

```
let f@x = lift (x +@ 1) in
  f@3 + f@4
```

which is just syntactic sugar for a `let` definition binding `f` to a static λ-expression `\@x....`

We can also make the `let` itself unfoldable, causing the specialiser to substitute the specialised first expression into the second, rather than build a residual `let` expression. For example,

```
let@ f x = lift (x +@ 1) in
  f 3 + f 4
```

But note that making a `let` expression unfoldable *has no effect* on the static information available during specialisation; the residual type of the bound variable is the same regardless of whether or not its specialised code is substituted into the body. In this example, even though the `let` is unfolded, `f` is still bound to a dynamic λ-expression, and so only *one* specialisation of `f` is generated to be substituted for each occurrence. Of course, as a result we encounter the 'cannot unify `3` and `4`' error.

The syntactic sugar extends to polyvariant function definitions: the polyvariant version of our example can be written

```
let poly f x = lift (x +@ 1) in
  spec f 3 + spec f 4
```

which is sugar for

```
let f = poly \x. ... in ...
```

Finally, recursive functions can be defined with `letrec`; to make a recursive function unfoldable, label both the `letrec` and the corresponding λs static[1].

5 *Exercise*: Specialising the `power` function.

Now we can set our first exercise: the `power` function (with a static first parameter) can be defined and tested by

[1] There is a trap for the unwary here. As far as the specialiser is concerned, it makes perfect sense to define an unfoldable recursive function using a dynamic `letrec`. The effect is to unfold calls, but generate a recursive definition of the residual part of the function value. Although this is unproblematic for the specialiser, such recursive definitions would usually cause the arity raiser to construct a type with an infinite representation, and this is reported as an error. To avoid these rather obscure errors, see to it that you always define recursive unfoldable functions with `letrec@` rather than `letrec`.

```
letrec power n x = if@ n=@ 1 then x else x*power (n-@1) x
in \x. power 3 x
```

But this program cannot be specialised as it stands, because `n` would need to be assigned several different residual types.

- Solve the problem by making the `power` function polyvariant. Notice the effect of arity raising.
- Alternatively, make the `power` function unfoldable instead.
- Find a third solution, by using constructor specialisation to pass several different static arguments to `power`.

6 Datatypes in the Type Specialiser

The type specialiser supports datatypes with constructors, with pattern matching over them in `case` expressions, much as in Haskell [2] (although nested patterns are not allowed). Any name beginning with a capital letter is a constructor. However, datatypes are not declared: instead, appropriate type definitions (recursive if need be) are inferred automatically by the type specialiser. Moreover, the same constructor can appear in more than one inferred datatype. The same holds true of residual types.

For example,

- `Pair (lift 1) (lift 2)`
  specialises to `Pair 1 2` with residual type `Pair int int`. We reuse the constructor name to express the type it constructs.
- `\b. if b then Left (lift 1) else Right (lift 2)`
  specialises to `\b.if b then Left 1 else Right 2` with residual type

  ```
  bool->Left int | Right int
  ```

  which illustrates how we write types with more than one constructor.
- `Cons (lift 1) (Cons (lift 2) Nil)`
  specialises to `Cons 1 (Cons 2 Nil)`. Here the two occurrences of `Cons` and the occurrence of `Nil` are quite independent; nothing forces them to belong to the same type, and so the specialiser infers the residual type of this term to be `Cons int (Cons int Nil)`.
- `letrec x = Cons (lift 1) x in x`
  specialises to `letrec x = Cons 1 x in x`, with a recursive residual type which can be expressed as

  ```
  _5 where _5 = Cons int _5
  ```

  (Here _5 is a type variable).

The reason for this slightly odd language design is that the specialiser *must* be able to infer new specialised type definitions in residual programs, since the programmer obviously cannot write them, and it is convenient then to allow constructors to appear in many types, so that the specialiser is not forced to invent new constructor names. Given that we infer appropriate data types in residual programs, it then seems natural to do the same in the source language.

Dynamic data structures may very well have static components: recall once again that the type specialiser does *not* restrict where static values may appear. For example, `Pair 1 (lift 2)` specialises to `Pair void 2` with residual type `Pair 1 int`. Arity raising then erases the void component, so that the final term is just `Pair 2`.

As usual, the static information is carried by the residual type. Of course, for residual types to match, this means that *every* `Pair` that an expression might evaluate to must have the same static component. For example,

```
\b. if b then Pair 1 (lift 2) else Pair 2 (lift 3)
```

cannot be specialised: if we try, we get a 'cannot unify `1` with `2`' error.

However, if the two arms of the conditional use *different* constructors, then the program becomes specialisable. For example,

```
\b. if b then Left 2 else Right 3
```

specialises (after arity raising) to

```
\b.if b then Left else Right
```

with residual type `bool->Left 2 | Right 3`. Such a type carries just the right information to specialise a `case` expression; we specialise the `Left` branch to `2`, and the `Right` branch to `3`.

So why not allow residual types such as `Pair 1 int | Pair 2 int`, so that the previous example could also be specialised? The problem with such types is that the constructor name is no longer sufficient to identify the residual types of the components. A `case` over this type would need *two* specialised `Pair` branches, and of course we cannot allow two branches with the same constructor. To make this idea work, we would need to generate two specialised versions of the `Pair` constructor. But constructor specialisation is already provided via the special constructor `In`, as we have already seen. There is no need to provide it in the context of ordinary datatypes also.

Data types may also have static constructors. In this case, the constructors appear in residual types, not in residual terms. So for example, `Left@ (lift 1)` specialises to just `1`, in which no constructor appears. But the residual type is `Left@ int`, recording the fact that the value represents an application of `Left@`. Since the residual type of a static data type tells us exactly which constructor was applied, then `case@` expressions over such types can always be specialised to one of their branches. For example,

```
let p = Pair@ (lift 1) (lift 2) in
let first x = case@ x of Pair y z: y esac in
first p
```

specialises to

```
let p_1 = 1
    p_2 = 2
in let first x_1 x_2 = x_1
   in first p_1 p_2
```

from which both the `Pair@` constructor, and the `case@` over it, have been removed. Notice that removing the constructor exposes the tuple of its components, which the arity raiser then removes.

7 *Exercise*: Specialising the `append` function.

The `append` function can be defined and tested as follows:

```
letrec append xs ys =
  case xs of
    Nil: ys,
    Cons x xs: Cons x (append xs ys)
  esac
in
append (Cons (lift 1) (Cons (lift 2) Nil)) (Cons (lift 3) Nil)
```

This program can be supplied to the specialiser as it stands, but no interesting specialisation occurs since all the constructors are dynamic.

- Make the constructors of the *first* argument static, to create a specialised version of `append` for lists of length two. You will need to do more than simply change the binding times.
- Now make the constructors of the second argument and of the result static also. Can you explain the result you obtain?

8 *Exercise*: Experiments with Arity Raising.

We can explore the behaviour of the arity raiser by giving the specialiser purely dynamic terms with pairs in different contexts, and examining the output. For terms with no static parts, the type specialisation phase is the identity function, and so we know that the term we write is the input to the arity raiser. Thus we can explore its behaviour in isolation.

For example, try specialising

```
let f x = x in f (lift 1, lift 2)
```

and

```
let proj x = case x of Inj y: y esac
in proj (Inj (lift 1, lift 2))
```

You will observe duplication of code in the output. Can you find a simple modification to the examples which avoids it? *Hint: insert a dynamic constructor somewhere.*

To see a larger example of arity raising at work, try defining function composition as a higher order function, and a function `swap` which swaps the components of a pair, and then try composing `swap` with itself.

9 *Exercise*: Simulating Constructor Specialisation: An Exercise in First-Class Polyvariance.

Polyvariance in the type specialiser is more general than in most partial evaluators, in that *any* expression can be specialised polyvariantly, not just top-level definitions. In this exercise we will show that such first-class polyvariance can actually simulate constructor specialisation. To do so, we shall model the specialisable constructor `In`, and `case` over it, by λ-terms which are specialised polyvariantly.

One way to model data-structures in the pure lambda-calculus is as follows. Model data values by functions from the branches of a `case` over the datatype to the result of the `case`. Model `case` branches by functions from the components to the result of the branch. Model constructors by functions which select the appropriate branch, and apply it to the components. For example, lists can be modelled as follows:

```
nil c n = n
cons x xs c n = c x xs
listcase xs c n = xs c n
```

If we apply this idea to a datatype with one unary constructor, we obtain

```
inj x k = k x
caseinj x k = x k
```

Now we want to model the behaviour of the *specialisable* constructor `In`. By representing `case` branches as a *polyvariant* function, show how `poly` and `spec` can model constructor specialisation.

10 *Exercise*: Optimal Specialisation.

A self-interpreter for the specialiser's metalanguage can be specialised optimally if, for any program p,

$$mix\ int\ p = p$$

(up to trivialities such as renaming of variables). Intuitively, the specialiser can remove a complete layer of interpretation. If the meta-language is typed, then an optimal specialiser must specialise types, since otherwise this equation cannot hold for any p containing a type not found in *int*. In particular, the 'universal' type used to represent values in the interpreter must be specialised to the types of those values. The type specialiser was the first to be able to do so for the lambda-calculus. In this exercise, you will repeat this experiment.

Take the following typed interpreter for the lambda-calculus plus constants:

```
letrec eval env e =
  case e of
    Cn n: Num n,
    Vr x: env x,
    Lm x e: Fun (\v.
             let env y = if x=y then v else env y
             in eval env e),
    Ap e1 e2:case eval env e1 of
               Fun f: f (eval env e2)
             esac
  esac
in

eval (\i.Wrong)

(Ap (Ap (Lm (lift "x") (Lm (lift "f")
          (Ap (Vr (lift "f")) (Vr (lift "x"))))) (Cn (lift 3)))
    (Lm (lift "z") (Vr (lift "z"))))
```

For your convenience, it is applied to a small example program.

The interpreter can be specialised as it stands, but since everything is dynamic then no specialisation occurs. Make the following changes to the binding-times in the interpreter, along with any other necessary changes to make specialisation possible, and see how the results change.

- Make the constructors `Cn`, `Vr`, `Lm` and `Ap` static.
- Make the constants and variable names in the program static.
- Unfold calls of `eval`, if you have not already done so.
- Make the constructors `Num`, `Fun` and `Wrong` static.

Have you achieved optimal specialisation? (If not: keep trying.) What happens if you specialise this interpreter to an ill-typed lambda-term, such as

```
(Ap@ (Cn@ 3) (Cn@ 4))
```

Is this the behaviour you would expect?

11 *Exercise*: Transforming Polymorphism to Monomorphism.

The type specialiser is not optimal for the polymorphic λ-calculus, because both source and residual programs are simply typed (*i.e.* monomorphic). However, we can write an *interpreter* for a polymorphic language in the type specialiser's meta-language. Specialising such an interpreter to a polymorphic program will translate it into a monomorphic one.

Begin by adding a case to your optimal interpreter from the previous exercise so that it also interprets a **let** construct:

```
Let x e1 e2
```

represents **let** $x = e_1$ **in** e_2. Test your interpreter by specialising it to

```
(Let@ "id" (Lm@ "x" (Vr@ "x")) (Ap@ (Vr@ "id") (Cn@ 3)))
```

Make sure that specialisation is still optimal — that is, you obtain a corresponding **let** in the residual program.

What happens if you specialise your interpreter to a program which requires polymorphism to be well-typed? For example,

```
(Let@ "id" (Lm@ "x" (Vr@ "x"))
   (Ap@ (Ap@ (Vr@ "id") (Vr@ "id")) (Cn@ 3)))
```

Modify your interpreter so that it can be specialised to this term. You will need to generate *two* versions of `id` in the residual program, with two different monotypes — could polyvariance be useful perhaps? Following the Hindley-Milner type system, you may wish to distinguish between λ-bound and **let**-bound variables, where only the latter may be polymorphic.

12 *Exercise*: Transforming Higher-Order to First-Order.

Higher-order programs can be transformed to first-order ones by representing function values as data-structures called closures, consisting of a tag identifying the function, and the values of its free variables. Function calls are interpreted by calling a dispatch function which inspects the tag, and then behaves like the function that the tag identifies. The transformation to first-order form is called *closure conversion* or *firstification*, and is a little tricky in a typed language. The object of this exercise is to develop an interpreter for the λ-calculus, which when specialised to a λ-term produces the result of firstifying it.

Start from the optimal interpreter you developed above. Can you change the representation of functions in the interpreter in such a way that *residual* functions will be represented by tagged tuples of their free variables? Don't forget to introduce a dispatching function, which can be specialised to produce the dispatch function in the firstified code!

A suitable lambda-expression to test your firstifier on is

```
(Ap@ (Lm@ "ap" (Ap@ (Ap@ (Vr@ "ap") (Lm@ "z" (Vr@ "z")))
                    (Ap@ (Ap@ (Vr@ "ap") (Lm@ "w" (Cn@ 3)))
                         (Cn@ 4))))
     (Lm@ "f" (Lm@ "x" (Ap@ (Vr@ "f") (Vr@ "x")))))
```

which represents

$$(\lambda ap.ap\ (\lambda z.z)\ (ap\ (\lambda w.3)\ 4))\ (\lambda f.\lambda x.f\ x)$$

13 *Exercise*: Interpreting Imperative Programs.

Below is an interpreter for a simple imperative language, supporting assignments, conditional statements, and sequencing. Variables in the interpreted language need not be declared: a variable is given a value simply by assigning to it. The interpreter given below is purely dynamic; your job is to modify it so that the program to be interpreted, and the names in the environment, are static.

```
let look env x = env x in
let assign env x v =
   \i. if i=x then v else env i
in

letrec eval env e =
  case e of
    Con n: n,
    Var s: look env s,
    Add e1 e2: eval env e1 + eval env e2
  esac
in

letrec exec env p =
  case p of
    Skip: env,
    Assign x e: assign env x (eval env e),
    Seq p1 p2: let env=exec env p1 in exec env p2,
    If e p1 p2: if eval env e=lift 0 then exec env p2
                else exec env p1
  esac
in

let run p e = let env = exec (\i.lift 0) p in
              eval env e
in

run
(Seq (Assign (lift "x") (Con (lift 3)))
     (Seq (If (Var (lift "x"))
              (Assign (lift "y")
                 (Add (Var (lift "x")) (Con (lift 1))))
              Skip)
          (Assign (lift "z") (Var (lift "y")))))

(Add (Var (lift "x")) (Var (lift "y")))
```

This interpreter would be difficult to specialise with a partial evaluator, because of the dynamic conditional in the function `exec`, which forces the result of `exec` to be dynamic. But `exec` returns the environment, which should of course be partially static. Luckily the type specialiser allows dynamic conditionals to have partially static results, so the problem will not arise. One solution using a partial evaluator would be to use CPS specialisation, which specialises the context of a dynamic conditional with partially static branches twice, once in each branch. In the example above, the statement following the `If` statement would need to be 'compiled' twice (that is, two different specialisations of `exec` to this statement would need to be generated), since one branch of the `If` introduces the variable `y` into the environment, while the other branch does not. Thus the partially static environment would have a different shape, depending on which branch of the conditional statement was chosen, and the reference to `y` in

```
(Assign (lift "z") (Var (lift "y")))
```

would need to be compiled differently in each case. The type specialiser on the other hand 'compiles' this last statement once only. Inspect the residual code: how is the problem of different variables in the environment after each branch of an `If` resolved?

In the interpreter given above, uninitialised variables have the value zero. Modify the interpreter to distinguish between initialised and uninitialised variables in the environment. What is the effect of making this distinction static, in the example above?

(The answer to this exercise is not included in the appendix).

References

[1] D. Dussart, J. Hughes, and P. Thiemann. Type Specialisation for Imperative Languages. In *International Conference on Functional Programming*, Amsterdam, June 1997. ACM.

[2] P. Hudak, S. L. Peyton Jones, and P. Wadler (editors). Report on the programming language haskell, a non-strict purely functional language (version 1.2). *SIGPLAN Notices*, Mar, 1992.

[3] J. Hughes. An Introduction to Program Specialisation by Type Inference. In *Functional Programming.* Glasgow University, July 1996. published electronically.

[4] J. Hughes. Type Specialisation for the Lambda-calculus; or, A New Paradigm for Partial Evaluation based on Type Inference. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, volume 1110 of *LNCS*. Springer-Verlag, February 1996.

[5] J. Hughes. Type Specialisation. *Computing Surveys*, to appear, 1998.

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

A Answers to the Exercises.

A.1 Specialising the `power` function.

To make `power` *polyvariant:*

```
letrec poly power n x = if@ n=@ 1 then x else x*spec power (n-@1) x
in \x. spec power 3 x
```

producing residual code

```
letrec power_1 x' = x' * power_2 x'
       power_2 x' = x' * power_3 x'
       power_3 x' = x'
in \x.power_1 x
```

In this example, the *specialiser* produces a single recursive definition of a 3-tuple from the recursive definition of `power`, and it is the *arity raiser* which splits this into three separate function definitions.

To unfold the `power` *function:*

```
letrec@ power@n@x = if@ n=@ 1 then x else x*power@(n-@1)@x
in \x. power@3@x
```

producing residual code

```
\x.x * (x * x)
```

To use constructor specialisation:

```
letrec power m x =
  case m of In n:
    if@ n=@ 1 then x else x*power (In (n-@1)) x
  esac
in \x. power (In 3) x
```

producing residual code

```
letrec power m x =
         case m of
           In0 : x,
           In1 : x * power In0 x,
           In2 : x * power In1 x
         esac
in \x.power In2 x
```

A.2 Specialising the `append` function.

To change the binding times of the constructors in the first argument, we annotate the `case@` in `append` static, and similarly for the constructors in the actual parameter.

```
letrec append xs ys =
  case@ xs of
    Nil: ys,
    Cons x xs: Cons x (append xs ys)
  esac
in
append (Cons@ (lift 1) (Cons@ (lift 2) Nil@)) (Cons (lift 3) Nil)
```

But this program cannot be specialised as it stands: we obtain the error message 'Cannot unify `Nil@` with `Cons@ int (Nil@)`'. The problem is that a specialisation of `append` to lists of length two needs to call a specialisation to lists of length one, and so on. Thus we must specialise the body of `append` several times.

This is the same problem that we encountered in the previous example, and we can use the same solutions. Using polyvariance, we rewrite the program as

```
letrec poly append xs ys =
  case@ xs of
    Nil: ys,
    Cons x xs: Cons x (spec append xs ys)
  esac
in
spec append (Cons@ (lift 1) (Cons@ (lift 2) Nil@))
            (Cons (lift 3) Nil)
```

which specialises to

```
letrec append_1 xs_1 xs_2 ys = Cons xs_1 (append_2 xs_2 ys)
       append_2 xs ys = Cons xs (append_3 ys)
       append_3 ys = ys
in append_1 1 2 (Cons 3 Nil)
```

Notice that once the static constructors of `append`'s first argument are removed, then the arity raiser can replace it by several arguments, one for each component.

Alternatively, we might use constructor specialisation, and tag `append`'s first argument with the specialisable constructor `In`:

```
letrec append t ys =
  case t of
    In xs:
      case@ xs of
        Nil: ys,
        Cons x xs: Cons x (append (In xs) ys)
      esac
  esac
in
append (In (Cons@ (lift 1) (Cons@ (lift 2) Nil@)))
       (Cons (lift 3) Nil)
```

In this case, the residual program is

```
letrec append t ys =
         case t of
           In0 : ys,
           In1 xs: Cons xs (append In0 ys),
           In2 xs_1 xs_2: Cons xs_1 (append (In1 xs_2) ys)
         esac
in append (In2 1 2) (Cons 3 Nil)
```

Here the specialised constructors provide efficient representations for lists of length zero, one and two, and the `append` function contains a `case` dispatch that enables it to append any one of these to another list.

Taking the polyvariant solution as a starting point, we can make the other constructors static also just by annotating them with a `@`; no other changes are required.

```
letrec poly append xs ys =
  case@ xs of
    Nil: ys,
    Cons x xs: Cons@ x (spec append xs ys)
  esac
in
spec append (Cons@ (lift 1) (Cons@ (lift 2) Nil@))
            (Cons@ (lift 3) Nil@)
```

But the residual program we obtain is surprisingly large; it is:

```
(letrec append_1 xs_1 xs_2 ys = xs_1
        append_2 xs_1 xs_2 ys = append_4 xs_2 ys
        append_3 xs_1 xs_2 ys = append_5 xs_2 ys
        append_4 xs ys = xs
        append_5 xs ys = append_6 ys
        append_6 ys = ys
 in append_1 1 2 3,
 (letrec append_1 xs_1 xs_2 ys = xs_1
         append_2 xs_1 xs_2 ys = append_4 xs_2 ys
         append_3 xs_1 xs_2 ys = append_5 xs_2 ys
         append_4 xs ys = xs
         append_5 xs ys = append_6 ys
         append_6 ys = ys
  in append_2 1 2 3,
  letrec append_1 xs_1 xs_2 ys = xs_1
         append_2 xs_1 xs_2 ys = append_4 xs_2 ys
         append_3 xs_1 xs_2 ys = append_5 xs_2 ys
         append_4 xs ys = xs
         append_5 xs ys = append_6 ys
         append_6 ys = ys
  in append_3 1 2 3))
```

with residual type `Cons@ int (Cons@ int (Cons@ int (Nil@)))`.

This large term is the result of arity raising. After just the type specialisation phase, the residual term is more recognisable:

```
letrec append = (\xs.\ys. (fst xs, pi2 append (snd xs) ys),
                 \xs.\ys. (fst xs, pi3 append (snd xs) ys),
                 \xs.\ys. ys)
in pi1 append (1,(2,void)) (3,void)
```

Here the `append` function has been replaced by a triple of specialisations, while the static list constructors have been removed. The residual lists are just represented by nested pairs of their dynamic components.

Arity raising eliminates the triple that `append` is bound to by transforming it into separate function definitions instead. It eliminates the tuples passed to `append` by passing the components as separate parameters instead. But it also eliminates the tuples in the *result* of `append`, by splitting each function that returns a tuple into several functions, one returning each component. Since the first specialisation of `append` returns nested pairs with a total of three components, then it is split into three separate functions, `append_1`, `append_2`, and `append_3` (one for each list element in the result of the original call). The second specialisation of `append` returns a structure with two integer components, and is split into `append_4` and `append_5`. Finally, the last specialisation corresponds to `append_6`.

This process results in an expression of the form `letrec...in (a,(b,c))`. In order to eliminate the last pairs, the arity raiser moves the `letrec` definitions into each component, thus transforming the entire expression into a triple of independent expressions, within which no tuples appear. It is this last step which creates multiple copies of the residual function definitions.

We will explore the arity raiser's behaviour further in the next exercise.

A.3 Experiments with Arity Raising.

Specialising the first example yields

```
(let f_1 x_1 x_2 = x_1
     f_2 x_1 x_2 = x_2
 in f_1 1 2,
 let f_1 x_1 x_2 = x_1
     f_2 x_1 x_2 = x_2
 in f_2 1 2)
```

Just as in the previous exercise, the residual `let` definition is floated inside the pair, in order to express the entire term in the form `(a,b)`. To prevent this, we must modify the body of the `let` expression, so that it no longer has a product type. A simple way to do so is to wrap a dynamic constructor around the body:

```
let f x = x in Wrap (f (lift 1, lift 2))
```

Specialising this expression produces

```
let f_1 x_1 x_2 = x_1
    f_2 x_1 x_2 = x_2
in Wrap (f_1 1 2) (f_2 1 2)
```

Now the arity raiser eliminates the pair in the body of the `let` just by giving the `Wrap` constructor extra components; the definitions of `f_1` and `f_2` need no longer be duplicated.

The code duplication in the second example, and indeed in the `append` example in the previous exercise, can be avoided in the same way.

We can define and test `compose` and `swap` as follows:

```
let compose f g x = f (g x) in
let swap x = case@ x of Pair y z: Pair@ z y esac in
let h = compose swap swap in
Wrap (h (Pair@ (lift 1) (lift 2)))
```

Here values constructed by `Pair@` specialise to pairs, which the arity raiser then removes; we use these values rather than the explicit pair notation because the implemented type specialiser does not support pattern matching on the latter.

The result of specialising this example is

```
let compose_1 f_1 f_2 g_1 g_2 x_1 x_2 =
      f_1 (g_1 x_1 x_2) (g_2 x_1 x_2)
    compose_2 f_1 f_2 g_1 g_2 x_1 x_2 =
      f_2 (g_1 x_1 x_2) (g_2 x_1 x_2)
in let swap_1 x_1 x_2 = x_2
       swap_2 x_1 x_2 = x_1
   in let h_1 = compose_1 swap_1 swap_2 swap_1 swap_2
          h_2 = compose_2 swap_1 swap_2 swap_1 swap_2
      in Wrap (h_1 1 2) (h_2 1 2)
```

Here `swap` is a function from pairs to pairs, and so is arity raised into two functions, each with two arguments. Thus `compose`, which takes two such functions and a pair as arguments, becomes a function of six parameters after arity raising – or rather two functions of six parameters, since it also returns a pair.

A.4 Simulating Constructor Specialisation: An Exercise in First-Class Polyvariance

Recall our first example of constructor specialisation:

```
(\f. f (In 3) + f (In 4))
  (\z. case z of In x: lift (x +@ 1) esac)
```

Let us use this example to test our simulation using polyvariance. To begin with, let us replace `In` and the `case` by the functions given in the statement of the exercise, that simulate a single unary constructor:

```
let inj x k = k x in
let caseinj x k = x k in
(\f. f (inj 3) + f (inj 4))
  (\z. caseinj z (\x. lift (x +@ 1)))
```

This cannot be specialised as it stands, of course, because `inj` is applied to both `3` and `4`, which causes a unification failure. We have to make `inj` *polyvariant*, so that it can specialise to many different 'constructors':

```
let poly inj x k = k x in
let caseinj x k = x k in
(\f. f (spec inj 3) + f (spec inj 4))
  (\z. caseinj z (\x. lift (x +@ 1)))
```

However, attempting to specialise this program still produces an error message, 'cannot unify `3` with `4`'. The problem is caused by the λ-expression passed to `caseinj`, the 'body of the `case`', which must also be specialised to `3` and `4`. We therefore make this λ-expression polyvariant where it is passed to `caseinj`, and insert a corresponding `spec` in the body of `inj`, where the 'body of the `case`' is finally invoked. The result is

```
let poly inj x k = spec k x in
let caseinj x k = x k in
(\f. f (spec inj 3) + f (spec inj 4))
  (\z. caseinj z (poly \x. lift (x +@ 1)))
```

Specialisation now succeeds, and produces

```
let inj_1 k_1 k_2 = k_2
    inj_2 k_1 k_2 = k_1
in let caseinj x k_1 k_2 = x k_1 k_2
   in (\f.f inj_1 + f inj_2) (\z.caseinj z 5 4)
```

We can see that the two functions `inj_1` and `inj_2` model the constructors of a type with two nullary constructors, and `caseinj` models a case over such a type. At the call of `caseinj`, two specialised '`case` branches' are passed as parameters. This program is exactly the λ-calculus model of the program we obtain by specialising the original code which uses built-in constructor specialisation:

```
(\f.f In1 + f In0) (\z.case z of In0 : 5, In1 : 4 esac)
```

Of course, if there were more than one call of `caseinj` then we might wish to generate several residual versions of it; we ought to make `caseinj` polyvariant also to allow for this. But even so, we cannot hope to model *every* use of constructor specialisation by this technique. The problem is that the simple type system of our source language constrains us too hard: because it does not support polymorphism, the `inj` function can only be applied to arguments of a single (source) type. Indeed, to model datatypes fully in a typed λ-calculus requires first-class polymorphism. What this exercise shows is that, if the type specialiser could be extended to handle a language with first-class polymorphism, then constructor specialisation could be reduced to polyvariance. In the meantime it must be provided as a separate feature.

A.5 Optimal Specialisation

Specialising the interpreter as it is given produces a residual program identical up to renaming:

```
letrec eval env e =
         case e of
           Cn n: Num n,
           Vr x: env x,
           Lm xe_1 xe_2: Fun (\v.let env' y =
                                       if xe_1 = y then v else env y
                                 in eval env' xe_2),
           Ap e1e2_1 e1e2_2: case eval env e1e2_1 of
                               Fun f: f (eval env e1e2_2)
                             esac
         esac
in eval (\i.Wrong)
     (Ap (Ap (Lm "x" (Lm "f" (Ap (Vr "f") (Vr "x")))) (Cn 3))
         (Lm "z" (Vr "z")))
```

The only difference from the source program is the names of `case`-bound variables: as you see, the type specialiser fails to preserve their names properly. Of course, this is just a deficiency of my particular implementation, and nothing fundamental.

The residual type reported may look a little surprising at first sight:

```
Residual type: Num _12 | _13
where
_13 = Fun (Num _12 | _13->Num _12 | _13) | Wrong
_12 = int
```

but it is isomorphic to

```
Univ
where
Univ = Num int | Fun (Univ->Univ) | Wrong
```

which is the type we would expect. The type specialiser represents types as graphs, and does not distinguish types which have the same infinite unfolding. Types are printed as *some* graph with the right infinite unfolding, but not always the one we would expect!

To make the constructors of the program syntax static, we annotate the `case` in `eval` as static, and likewise each occurrence of a constructor. But if we do no more than this, then we encounter a specialisation time error:

```
Error: Cannot unify:
Lm@ string (Lm@ string (Ap@ (Vr@ string) (Vr@ string)))

with
Ap@ (Lm@ string (Lm@ string (Ap@ (Vr@ string) (Vr@ string))))
    (Cn@ int)
```

These are the residual types of the arguments to the top-level and first recursive call of `eval`. Since the arguments of `eval` are now partially static, we must make it polyvariant. Doing so, we obtain

```
letrec poly eval env e =
  case@ e of
    ...
    Lm x e: Fun (\v.
             let env y = if x=y then v else env y
             in spec eval env e),
    Ap e1 e2:case spec eval env e1 of
               Fun f: f (spec eval env e2)
             esac
  esac
in

spec eval (\i.Wrong) ...
```

which specialises to

```
letrec eval_1 env e_1 e_2 e_3 e_4 e_5 e_6 e_7 =
         case eval_2 env e_1 e_2 e_3 e_4 e_5 of
           Fun f: f (eval_3 env e_6 e_7)
         esac
       eval_2 env e_1 e_2 e_3 e_4 e_5 =
         case eval_4 env e_1 e_2 e_3 e_4 of
           Fun f: f (eval_5 env e_5)
         esac
       eval_3 env e_1 e_2 =
         Fun (\v.let env' y = if e_1 = y then v else env y
                 in eval_6 env' e_2)
       eval_4 env e_1 e_2 e_3 e_4 =
         Fun (\v.let env' y = if e_1 = y then v else env y
                 in eval_7 env' e_2 e_3 e_4)
       eval_5 env e = Num e
       eval_6 env e = env e
       eval_7 env e_1 e_2 e_3 =
         Fun (\v.let env' y = if e_1 = y then v else env y
                 in eval_8 env' e_2 e_3)
       eval_8 env e_1 e_2 =
         case eval_6 env e_1 of Fun f: f (eval_6 env e_2) esac
in eval_1 (\i.Wrong) "x" "f" "f" "x" 3 "z" "z"
```

Notice all the arguments to the residual versions of `eval`! Inspection of the final call reveals that they are the identifiers and constants in the interpreted program; of course, we must make these static also. The easiest way to do so is to remove the uses of `lift` from the term to interpret, and instead `lift` the constants and variables when they are used in `eval`. We obtain

```
letrec poly eval env e =
  case@ e of
    Cn n: Num (lift n),
    Vr x: env (lift x),
    Lm x e: Fun (\v.
             let env y = if lift x=y then v else env y
             in spec eval env e),
    ...
  esac
in

spec eval (\i.Wrong)

(Ap@ (Ap@ (Lm@ "x" (Lm@ "f"
             (Ap@ (Vr@ "f") (Vr@ "x")))) (Cn@ 3))
     (Lm@ "z" (Vr@ "z")))
```

Specialising this program, we obtain

```
letrec eval_1 env =
         case eval_2 env of Fun f: f (eval_3 env) esac
       eval_2 env = case eval_4 env of Fun f: f (eval_5 env) esac
       eval_3 env =
         Fun (\v.let env' y = if "z" = y then v else env y
                 in eval_6 env')
       eval_4 env =
         Fun (\v.let env' y = if "x" = y then v else env y
                 in eval_7 env')
       eval_5 env = Num 3
       eval_6 env = env "z"
       eval_7 env =
         Fun (\v.let env' y = if "f" = y then v else env y
                 in eval_8 env')
       eval_8 env = case eval_9 env of Fun f: f (eval_10 env) esac
       eval_9 env = env "f"
       eval_10 env = env "x"
in eval_1 (\i.Wrong)
```

However, in this version the environment lookup is still done dynamically. Of course, now that we have made the identifiers static, we would like to perform environment lookups statically also. The environment is a function from dynamic strings to values; we must change the type of its argument to static strings. Moreover, since one environment may very well be applied to several different static identifiers, the environment must be a *polyvariant* function. We therefore reannotate the source program as follows:

```
letrec poly eval env e =
  case@ e of
    ...
    Vr x: spec env x,
    Lm x e: Fun (\v.
             let poly env y = if@ x=@y then v else spec env y
             in spec eval env e),
    ...
  esac
in

spec eval (poly \i.Wrong) ...
```

and now obtain

```
letrec eval_1 = case eval_2 of Fun f: f eval_3 esac
       eval_2 = case eval_4 of Fun f: f eval_5 esac
       eval_3 = Fun (\v.let env' = v in eval_6 env')
       eval_4 = Fun (\v.let env' = v in eval_7 env')
       eval_5 = Num 3
       eval_6 env = env
       eval_7 env =
         Fun (\v.let env'_1 = v
                     env'_2 = env
                 in eval_8 env'_1 env'_2)
       eval_8 env_1 env_2 =
         case eval_9 env_1 env_2 of
           Fun f: f (eval_10 env_1 env_2)
         esac
       eval_9 env_1 env_2 = env_1
       eval_10 env_1 env_2 = env_2
in eval_1
```

as the result of specialisation.

Looking at this code, we see that the residual environment has been replaced by multiple parameters, one for each name in the environment. How has this happened? Firstly, since the environment is now a polyvariant object, its residual value is a tuple of specialisations, which the arity raiser then splits into individual parameters. There will be one specialisation (= one parameter) for each name *which is looked up* in the environment, not for each name which is bound in it. Although this example doesn't show it, polyvariance gives us 'dead variable elimination' for free.

Each component of the tuple is the specialisation of the environment to look up one particular identifier, and has residual type `x->Value`, where `x` is the identifier it looks up. Since identifiers are completely static, the arity raiser converts such functions just to values.

Thus the environment is replaced in residual programs by individual parameters, each of which is the value of a name in the interpreted program.

Up to this point, it is useful *not* to unfold calls of `eval`, because that enables us to see residual environments clearly. But of course, we cannot obtain optimal specialisation without unfolding `eval`, so let us do so now. We convert `eval` into a static function (and define it therefore with a static `letrec`), and since the body is now specialised at each call, it need no longer be polyvariant. The reannotated program is

```
letrec@ eval@ env@ e =
  case@ e of
    ...
    Lm x e: Fun (\v.
             let poly env y = if@ x=@y then v else spec env y
             in eval@ env@ e),
    Ap e1 e2:case eval@ env@ e1 of
               Fun f: f (eval@ env@ e2)
             esac
  esac
in

eval@ (poly \i.Wrong)@ ...
```

and the residual code is

```
case case Fun (\v.let env = v
                  in Fun (\v'.let env'_1 = v'
                                  env'_2 = env
                              in case env'_1 of
                                   Fun f: f env'_2
                                 esac))
     of
       Fun f: f (Num 3)
     esac
of
  Fun f: f (Fun (\v.let env = v in env))
esac
```

Now it is clear that the major remaining overhead is the tagging and tag checking of the constructors of the value type: `Fun`, `Num` and `Wrong`. Since we intend the *type* of interpreted values to be a compile-time property, we can make these constructors static also. By doing so, we exploit the extra power of the type specialiser: since function values in the interpreted language map values to values, and since such function values are necessarily dynamic, then most partial evaluators would insist that the value type itself must be completely dynamic. This is a consequence of the rule that a dynamic function may not have static arguments or results. But since the type specialiser places no such restrictions, we are free to reannotate the interpreter as

```
letrec@ eval@ env@ e =
  case@ e of
    Cn n: Num@ (lift n),
    Vr x: spec env x,
    Lm x e: Fun@ (\v.
             let poly env y = if@ x=@y then v else spec env y
             in eval@ env@ e),
    Ap e1 e2:case@ eval@ env@ e1 of
               Fun f: f (eval@ env@ e2)
             esac
  esac
in

eval@ (poly \i.Wrong@)@ ...
```

and obtain

```
(\v.let env = v
    in \v'.let env'_1 = v'
               env'_2 = env
           in env'_1 env'_2)
  3
  (\v.let env = v in env)
```

as the residual code. This is not quite isomorphic to the interpreted program, because the variables in the environment are rebound in every λ-expression. The residual `let` expressions in this code arise from the `let` in the interpreter in the case for `Lm x e`; replacing this `let` with `let@` we obtain instead

```
(\v.\v'.v' v) 3 (\v.v)
```

as the residual code, and we have attained optimal specialisation at last.

Notice that the residual type of this specialisation is now `Num@ int`; that is, we know statically that the result is an integer, representing a value tagged `Num@`. Specialising this interpreter to a λ-expression both 'compiles' it into the meta-language, and infers its type. If we try to specialise the interpreter to an ill-typed term, such as `(Ap@ (Cn@ 3) (Cn@ 4))`, then residual type inference fails:

```
Error: Cannot unify: Fun@ _20 with Num@ _26
```

We could hardly expect anything else: by making the constructors of the value type static, we 'compile' programs to code without run-time type checks; compilers for statically typed languages are able to do that precisely *because* they reject ill-typed programs altogether.

Another way to look at it is like this: an optimal specialiser must satisfy

$$mix\ int\ p = p$$

by definition; specialising a suitable interpreter to a program p must yield an identical program as the result (up to renaming, etc). But if the input program p is ill-typed, then the result of specialisation must also be ill-typed, by this equation. An optimal specialiser which generates well-typed residual programs in a typed language *must* therefore reject such inputs, because its defining equation cannot be satisfied at all.

A.6 Transforming Polymorphism to Monomorphism

Let us begin by adding a new case to the interpreter to handle `let` expressions:

```
letrec@ eval@ env@ e =
  case@ e of
    ...
    Let x e1 e2: let v = eval@ env@ e1 in
                 let@ poly newenv y =
                        if@ x=@y then v else spec env y
                 in eval@ newenv@ e2
  esac
in

eval@ (poly \i.Wrong@)@

(Let@ "id" (Lm@ "x" (Vr@ "x")) (Ap@ (Vr@ "id") (Cn@ 3)))
```

Notice that we bind `v` to the result of evaluating `e1` with a *dynamic* `let`; this generates a `let` expression in the residual program corresponding to the `Let` in the interpreted program, and so we obtain optimal specialisation. In this case, the residual program is

```
let v v = v in v 3
```

which is isomorphic to the interpreted one. (The choice of names is a little confusing here, since the specialiser renames variables only when necessary to avoid a name clash. There *are* two different variables in this residual term, but they are both called `v`).

If we try to specialise this interpreter to the second example term, we encounter an error:

```
Cannot unify Num@ int with Fun@ _46 where ...
```

The residual `let`-bound variable corresponding to `id` cannot be given a *monomorphic* residual type which matches both occurrences.

The solution to this problem is to give that variable a polyvariant type, and specialise it at each occurrence. Thus we modify the new environment created by interpreting a `Let` to bind the new variable to a polyvariant object. But now we encounter a typing problem in the source language: the environment must be well-typed in the source program, it cannot map some variables to polyvariant values and others to monovariant ones.

Our solution is to tag the result of looking up a name in the environment as either monovariant or polyvariant; that is, we give the environment the source type

```
poly (String@ -> Mono@ Univ | Poly@ (poly Univ))
```

Of course these tags are static: we know at compile-time which variables are `Let`-bound. We modify the interpreter accordingly, adding tags where variables are bound, and checking them where variables are interpreted:

```
letrec@ eval@ env@ e =
  case@ e of
    ...
    Vr x: case@ spec env x of
            Mono v: v,
            Poly v: spec v
          esac,
    Lm x e: Fun@ (\v.
             let@ poly env y = if@ x=@y then Mono@ v else spec env y
             in eval@ env@ e),
    ...
    Let x e1 e2: let poly v = eval@ env@ e1 in
                 let@ poly newenv y =
                   if@ x=@y then Poly@ v else spec env y
                 in eval@ newenv@ e2
  esac
in

eval@ (poly \i.Mono@ Wrong@)@

(Let@ "id" (Lm@ "x" (Vr@ "x"))
   (Ap@ (Ap@ (Vr@ "id") (Vr@ "id")) (Cn@ 3)))
```

Residual environments now contain a tuple of specialisations for each `Let`-bound variable, one for each residual (mono) type at which the variable is used. The residual type of an environment records, for each variable, whether it is mono- or poly-morphic, and for the polymorphic ones, which instances it is used at. After arity raising, polymorphic variables are replaced by a collection of variables, one for each instance. Our example specialises to

```
let v_1 v' = v'
    v_2 v' = v'
in v_1 v_2 3
```

Thus we obtain the well-known transformation of Hindley-Milner polymorphic programs to simply typed ones as the specialisation of an interpreter.

A.7 Transforming Higher-Order to First-Order

In the optimal interpreter we have already developed, functions in the interpreted language are represented as *dynamic* functions from values to values, and therefore specialise to functions in residual programs. To obtain first-order residual programs, we must change this representation. The obvious choice is to represent function values by *static* functions instead. In residual programs, static functions are replaced by tuples of their free variables, which is close to the representation of closures that we are aiming for.

However, if we simply change the interpretation of `Lm` to construct a static function, and change the interpretation of `Ap` to use static application, then the effect of specialising the interpreter will be to unfold all function applications at the point of call. Ineed, the whole point of static applications is that they are unfolded! Since our little interpreted language is non-recursive this would actually work, but is of course not the solution we are looking for.

To avoid unfolding function calls at the point of application, we introduce a `dispatch` function to 'apply' a function value; the intention is that each function in the interpreted program will be unfolded only once, inside the body of the residual `dispatch` function. Intuitively we 'compile' interpreted programs to one `dispatch` function which can apply any function value that the interpreted program can generate. With these modifications, the interpreter becomes:

```
let dispatch f x = f@x in
letrec@ eval@ env@ e =
  case@ e of
    ...
    Lm x e: Fun@ (\@v.
             let@ poly env y = if@ x=@y then v else spec env y
             in eval@ env@ e),
    Ap e1 e2:case@ eval@ env@ e1 of
               Fun f: dispatch f (eval@ env@ e2)
             esac
  esac
in ...
```

However, specialising this interpreter to our example produces a unification failure, which in retrospect is easy to understand: by making function values static, we make the definition of a function a part of its residual type, with the result that each function-valued variable can only ever be bound to one function. In particular, `dispatch` may only be applied to one function. But of course, `dispatch` is used to invoke *every* function, and since our example contains more than one, a unification failure results.

To avoid it, we tag function representations with the specialisable constructor `In`. As a result, a function-valued variable can now be bound to different functions, but in the residual program they will be tagged with different specialised constructors. That is, the representation of a function value in residual programs will be a tag identifying the particular function, and a tuple of its free variables – exactly the conventional representation of closures. The residual `dispatch` function will contain a `case` with a branch for every possible function tag, as we would expect. The modifications read:

```
let dispatch f x = case f of In g: g@x esac in
letrec@ eval@ env@ e =
  case@ e of
    ...
    Lm x e: Fun@ (In (\@v.
              let@ poly env y = if@ x=@y then v else spec env y
              in eval@ env@ e)),
    ...
  esac
in ...
```

However, although we have now chosen the right representation for functions, we *still* get a unification failure when we try to specialise the interpreter! The problem is that the residual `dispatch` function must not only be able to apply different functions, it must be able to apply functions *with different residual types.* One residual `dispatch` function cannot apply both functions of type `Int->Int` and functions of type `Bool->Bool`. We must therefore make `dispatch` polyvariant, so that a different residual `dispatch` function can be generated for each *type* of function in the interpreted program. The modifications read:

```
let poly dispatch f x = case f of In g: g@x esac in
letrec@ eval@ env@ e =
  case@ e of
    ...
    Ap e1 e2:case@ eval@ env@ e1 of
               Fun f: spec dispatch f (eval@ env@ e2)
             esac
  esac
in ...
```

and at last specialisation succeeds, producing

```
let dispatch_1 f x =
      case f of
        In0 g_1 g_2 g_3: g_3 (g_2 x (In1 x g_1 g_2 g_3))
                             (g_3 (g_2 x (In0 x g_1 g_2 g_3)) 4)
      esac
```

```
    dispatch_2 f x =
      case f of In0 g_1 g_2 g_3: In2 x g_1 g_2 g_3 esac
    dispatch_3 f x =
      case f of
        In0 g_1 g_2 g_3 g_4: 3,
        In1 g_1 g_2 g_3 g_4: x,
        In2 g_1 g_2 g_3 g_4: g_4 g_1 x
      esac
in dispatch_1 (In0 dispatch_1 dispatch_2 dispatch_3)
     (In0 dispatch_1 dispatch_2 dispatch_3)
```

Just as we would expect, the residual program contains three residual `dispatch` functions, which interpret functions of type `Int->Int`, `(Int->Int)->(Int->Int)`, and `((Int->Int)->(Int->Int))->Int` from bottom to top. There are three different functions of type `Int->Int`, while there is only one of each of the other two types.

However, we observe that the function closures have bewilderingly many free variables. Moreover, the residual program is *not* first-order: some of these free variables are themselves function values! Indeed, they are specialised versions of the `dispatch` function itself!

What is going on here is that the static functions representing function values contain calls to `dispatch`, which is therefore a free variable, and appears in the function's residual representation. To avoid this, we must represent function values by static functions which *do not* depend on `dispatch`. We can do so simply, by passing the `dispatch` function as an extra parameter when a function value is invoked; thus the static function itself need not contain any reference to it. Since `dispatch` is actually called from `eval`, we need to make `dispatch` a parameter of `eval` also. The modifications to the interpreter read:

```
letrec poly dispatch f x = case f of In g: g@ dispatch@ x esac in
letrec@ eval@ dispatch@ env@ e =
  case@ e of
    ...
    Lm x e: Fun@ (In (\@dispatch1. \@v.
             let@ poly env y = if@ x=@y then v else spec env y
             in eval@ dispatch1@ env@ e)),
    Ap e1 e2:case@ eval@ dispatch@ env@ e1 of
               Fun f: spec dispatch f (eval@ dispatch@ env@ e2)
             esac
  esac
in

eval@ dispatch@ (poly \i.Wrong@)@ ...
```

Notice that `dispatch` itself must now be recursive, so that it can pass its own value to closures being invoked. Of course, we would expect the residual dispatcher functions to be recursive if the program to be firstified is; this recursion in residual programs arises from the `letrec` we have just introduced.

Notice also that in the case for interpreting a `Lm`-expression there are *two* versions of `dispatch` in scope. We are careful to use `dispatch1` rather than `dispatch`, so as to avoid referring to a free variable of the static function we are constructing.

The result of specialising this program is now

```
letrec dispatch_1 f x =
         case f of
           In0 : dispatch_3 (dispatch_2 x (In1 x))
                   (dispatch_3 (dispatch_2 x (In0 x)) 4)
         esac
       dispatch_2 f x = case f of In0 : In2 x esac
       dispatch_3 f x =
         case f of In0 g: 3, In1 g: x, In2 g: dispatch_3 g x esac
in dispatch_1 In0 In0
```

which we can at last see is a firstified version of the original term,

$$(\lambda ap.ap\ (\lambda z.z)\ (ap\ (\lambda w.3)\ 4))\ (\lambda f.\lambda x.f\ x)$$

– At type `Int->Int`,
  - `In0 ap` represents $\lambda w.3$,
  - `In1 ap` represents $\lambda z.z$,
  - `In2 f` represents $\lambda x.fx$.

  (Notice that closure representations contain all variables in scope, whether they are actually used or not: there is room for a little further improvement here).
– At type `(Int->Int)->(Int->Int)`, `In0` represents $\lambda f.\lambda x.f\ x$.
– At type `((Int->Int)->(Int->Int))->Int`, `In0` represents

$$\lambda ap.ap\ (\lambda z.z)\ (ap\ (\lambda w.3)\ 4)$$