J. Functional Programming **10** (4): 409–415, July 2000. Printed in the United Kingdom

FUNCTIONAL PEARL
Do we need dependent types?

DANIEL FRIDLENDER
BRICS, Department of Computer Science, University of Aarhus, Århus, Denmark*
(*e-mail:* daniel@brics.dk)

MIA INDRIKA
Department of Computing Science, Chalmers University of Technology and Göteborg University, Göteborg, Sweden
(*e-mail:* indrika@cs.chalmers.se)

1 The problem

This pearl is about some functions whose definitions seem to require a language with dependent types. We describe a technique for defining them in Haskell or ML, which are languages without dependent types.

Consider, for example, the scheme defining `zipWith` in figure 1. When this scheme is instantiated with `n` equal to `1` we obtain the standard function `map`. In practice, other instances of the scheme are often useful as well.

Figure 1 cannot be used as a definition of a function in Haskell because of the ellipses '. . .'. More importantly, the type of `zipWith` is parameterized by `n`, which seems to indicate the need for dependent types. However, as mentioned above, Haskell does not allow dependent types.

The way the Haskell library (Peyton Jones and Hughes, 1999a; Peyton Jones and Hughes, 1999b) solves the problem is by providing a family of 8 (!) functions `zipWith0`, `zipWith1`, `zipWith2`, `zipWith3`, . . . , `zipWith7`, where the number in the name of the function indicates the value given to `n` when instantiating the scheme.[1] The programmer can of course extend this family with more instances if (s)he needs.

```
zipWith :: (a1 -> ... -> an -> b) -> [a1] -> ... -> [an] -> [b]
zipWith f (a1:as1) ... (an:asn) = f a1 ... an : zipWith f as1 ... asn
zipWith _    _     ...    _     = []
```

Fig. 1. Scheme for `zipWith`.

* **B**asic **R**esearch **in** **C**omputer **S**cience, Centre of the Danish National Research Foundation.
[1] In the Haskell library, `zipWith0`, `zipWith1` and `zipWith2` are called `repeat`, `map` and `zipWith` respectively.

This mechanical repetition of code is very unpleasant. The main benefit of Haskell and ML polymorphism is precisely the ability to define functions at an abstract level, allowing a high degree of program reusability. However, in the case of `zipWith` this is only partially achieved. Every member of the family of functions has a polymorphic type, which means that it can be used to 'zip' lists of integers, booleans or of values of any other type. However, their definitions are still not abstract enough, since one cannot reuse them with different number of arguments.

The kind of problem that we address here is how to define within the Hindley–Milner type system (the core of the type systems underlying Haskell and ML) a general version of `zipWith` that can be used with a variable number of arguments. We describe a technique that introduces *ad hoc* codings for natural numbers, which resemble numerals in λ-calculus. The same technique can be applied to other examples such as `liftM` – for which the Haskell library also provides families of functions – and the tautology function `taut`, which is considered a standard example of the expressive power of dependent types.

2 A preliminary solution

As a motivating example, suppose we want to 'zip' eight given lists `as1`, . . . , `as8` with a given 8-ary function `f` of appropriate type in Haskell. For the reasons mentioned above, we decline defining a new instance `zipWith8` of the scheme in figure 1. We use instead the function `zipWith7` from the Haskell library, and write

```
zipWith7 f as1 as2 as3 as4 as5 as6 as7 << as8
```

where `<<` is defined as follows:

```
(<<) :: [a -> b] -> [a] -> [b]
(f:fs) << (a:as) = f a : (fs << as)
  _    <<   _    = []
```

In effect, since `f` is 8-ary, `zipWith7 f as1 as2 as3 as4 as5 as6 as7` returns a list of functions, and the operator `<<` makes sure that each function on that list is applied to the corresponding argument in `as8`.

Thus there is no need to define `zipWith8`: one can just write as above in terms of the existing `zipWith7`. Similarly, there is no need to use `zipWith7`, since it can be replaced by an expression written in terms of `zipWith6` and `<<`. Iterating this process, and assuming that `<<` associates to the left, the expression above can be written as

```
repeat f << as1 << ... << as8
```

where `repeat` – Haskell's name for `zipWith0` – is a function returning a list that consists of infinitely many copies of its argument, i.e.:

```
repeat :: b -> [b]
repeat f = f : repeat f
```

In general 'zipping' `n` given lists `as1`, . . . , `asn` with a given `n`-ary function `f` of appropriate type can be written as

```
  repeat f << as1 << ... << asn                                    (1)
```

in Haskell.

Using expressions like `(1)` is already more flexible than implementing many different instances of the scheme. The disadvantage is that the partial application `zipWith8 f as1` would have to be expressed in the following clumsy form:

```
  \as2 ... as8 -> repeat f << as1 << ... << as8
```

The final expression that we propose in the next section will solve this problem.

3 Introducing numerals

Notice that expression `(1)` contains not only the lists `as1`, . . . , `asn` to be 'zipped', but also extra explicit information about how many the lists are, namely, an occurrence of the operator `<<` for each of them. This gives rise to introducing numerals.

We define the successor function `succ` as follows:

```
  succ :: ([b] -> c) -> [a -> b] -> [a] -> c
  succ = \n fs as -> n (fs << as)
```

This can be read in terms of continuations: given a continuation `n`, a list of functions `fs` and a list of arguments `as`, it applies each function in `fs` to the corresponding argument in `as` producing a list which is given to the continuation `n`.

The numeral `zero` is simply the identity function `id :: a -> a`, which in particular has type `[a] -> [a]`. The remaining numerals are obtained by iterating the successor function `succ` on `zero`:

```
  one = succ zero :: [a -> b] -> [a] -> [b]
  two = succ one  :: [a -> b -> c] -> [a] -> [b] -> [c]
```

In general, the numeral `n̄` corresponding to the number `n` has the following type:

```
  n̄ :: [a1 -> ... -> an -> b] -> [a1] -> ... -> [an] -> [b]
```

We now define `zipWith` as:

```
  zipWith :: ([a] -> b) -> a -> b
  zipWith n f = n (repeat f)
```

Thus, given a numeral `n̄`, `zipWith n̄` will have type

```
  zipWith n̄ :: (a1 -> ... -> an -> b) -> [a1] -> ... -> [an] -> [b]
```

which is exactly what we wanted. Expression `(1)` can finally be written:

```
  zipWith n̄ as1 ... asn                                            (2)
```

The definitions for `zipWith` are summarized in figure 2.

```
(<<) :: [a -> b] -> [a] -> [b]
(f:fs) << (a:as) = f a : (fs << as)
  _    <<   _    = []

succ :: ([b] -> c) -> [a -> b] -> [a] -> c
succ = \n fs as -> n (fs << as)

zero = id :: a -> a           -- in particular [a] -> [a]
one = succ zero :: [a -> b] -> [a] -> [b]
two = succ one  :: [a -> b -> c] -> [a] -> [b] -> [c]

n̄  :: [a1 -> ... -> an -> b] -> [a1] -> ... -> [an] -> [b]

zipWith :: ([a] -> b) -> a -> b
zipWith n f = n (repeat f)

zipWith n̄ :: (a1 -> ... -> an -> b) -> [a1] -> ... -> [an] -> [b]
```

Fig. 2. Numerals for `zipWith`.

4 The numerals in use

We can now revisit the motivating example from section 2. Assume that the numeral `seven` is defined in the library. In order to 'zip' eight given lists `as1`, . . . , `as8` with a given 8-ary function `f` of appropriate type, we can define

```
eight = succ seven
```

and write the expression:

```
zipWith eight f as1 as2 as3 as4 as5 as6 as7 as8
```

Defining `eight` is of course unnecessary; one may replace it by `(succ seven)` in the expression above.

The disadvantage mentioned in section 2 vanishes now because the equivalent to `zipWith8 f as1` becomes:

```
zipWith eight f as1
```

We can also show with an example how these numerals can be reused. Suppose that we want to define a general function `zap` given by the following scheme:

```
zap :: [(a1 -> ... -> an -> b)] -> [a1] -> ... -> [an] -> [b]
zap (f:fs) (a1:as1) ... (an:asn) =
                              f a1 ... an : zap fs as1 ... asn
zap   _        _    ...    _     = []
```

A definition using the numerals from figure 2 would be

```
zap n = zipWith (succ n) id
```

which is certainly more convenient than following the approach that the Haskell library used for `zipWith`, defining a new family of functions for `zap`.

```
succ :: (a -> Bool) -> (Bool -> a) -> Bool
succ = \n f -> n (f True) && n (f False)

zero = id :: a -> a           -- in particular Bool -> Bool
one = succ zero :: (Bool -> Bool) -> Bool
two = succ one  :: (Bool -> Bool -> Bool) -> Bool

n̄ :: (Bool -> ... -> Bool -> Bool) -> Bool

taut n = n
```

Fig. 3. Numerals for `taut`.

5 Other examples

The idea of introducing numerals can be applied in several other cases. For each of the functions `liftM`, `zip`, `unzip`, `curry` and `uncurry` one can define `zero` and `succ`, which will make possible to give generic definitions of them.

We illustrate this with a final toy example that is interesting, since it is one of the typical small examples of programs that one can write thanks to having dependent types (Nordström *et al.*, 1990).

We want to define a function `taut` and numerals `n̄` such that `taut n̄` has type

```
taut n̄  :: (Bool -> ... -> Bool -> Bool) -> Bool
```

and `taut n̄ p` determines whether `p` – which represents a Boolean expression of `n` variables – is a tautology or not. This is achieved by making the definitions shown in figure 3.

6 Conclusion

Inspired by the work presented in Danvy (1998), we considered several functions whose implementability was generally believed to require dependent types. We have shown that it is possible to define such functions without dependent types in an elegant way by introducing ad hoc numerals.

The main disadvantage of this solution is precisely that the numerals we define are too ad hoc. For each example we have to define special purpose numerals. The numerals are so specific that the main function in each case becomes very simple (as in `zipWith`) or completely trivial (as in `taut`). We are seeking for a way of exploiting Haskell's polymorphism and overloading to find generic numerals, that is, numerals that would work for all or several of the functions `zipWith`, `liftM`, `zip`, `unzip`, `curry`, `uncurry` and `taut`.

The reader may observe that polymorphism is already being exploited in this paper all along. In figure 2, for example, `zipWith` and `succ` are defined in order to be applied to numerals. However, the numerals `zero`, `one`, `two`, ... have all different types. Fortunately, the type of the first argument in the definitions of `zipWith` and `succ` is more general than all of them. Probably the case of `taut` (figure 3) makes the use of polymorphism most evident, since the successor function `succ` has a polymorphic type even though the numerals do not.

Even though we have used Haskell syntax and assumed lazy evaluation in some places (when using the function `repeat`), the idea of using numerals only requires Hindley–Milner type system. With minor changes one can obtain numerals to implement a general `zipWith` in strict languages such as ML.

From the computational point of view the use of numerals in the case of `zipWith` produce a small loss of performance. Evaluating expression `(1)` or `(2)` implies generating and consuming a number of intermediate lists. For this reason, the performance of `zipWith` $\overline{\texttt{n}}$ as defined in figure 2 is a bit worse than the performance of the corresponding instance of the scheme in figure 1. We have verified by hand that with the deforestation algorithm given in Wadler (1990) it is possible to remove the generation and consumption of intermediate lists. This means that if we had a compiler with the ability of performing more advanced deforestation than present day compilers, `zipWith` $\overline{\texttt{n}}$ would be as efficient as the corresponding instance of the scheme in figure 1.

All the functions considered here can be generically defined in languages with dependent types, like Cayenne (Augustsson, 1998). However, it is not clear to us that this would benefit in a more convenient notation for the functions defined here. In the current implementations of languages with dependent types, `zipWith` would need to have extra parameter(s) with type information which, to the programmer, might be harder to write than the numerals.

In languages for generic programming (Jeuring and Jansson, 1996), it is possible to define a more general version of `zipWith` than the one in the Haskell library, a `zipWith` which would work for arbitrary datatypes rather than only for lists. However, the existing proposals of languages for generic programming cannot express the idea of a variable number of arguments needed for defining `zipWith` as we want here. It is possible that future proposals will. However, from the current development (Hinze, 1999) it seems fair to expect that again – just as in languages with dependent types – it would be necessary for the programmer to provide extra typing information. On the other hand, our technique can also be applied in a language for generic programming to define numerals for a `zipWith` which would then work for arbitrary datatypes and for a variable number of arguments.

We have given a general definition of the function `taut` in Hindley–Milner type system using numerals. As pointed out to us by Thomas Hallgren, if that type system is extended with classes like in Haskell then it is also possible to give a more convenient definition of `taut` which would not need numerals. That definition is obtained by overloading the function `taut`.

Acknowledgements

We are grateful to Magnus Carlsson and Olivier Danvy with whom we discussed the subject of this paper in several opportunities. Richard Bird, Olivier Danvy and an anonymous reviewer gave us valuable comments on earlier versions of this paper.

References

Augustsson, L. (1998) Cayenne – A Language with Dependent Types. *Proc. Third ACM SIGPLAN International Conference on Functional Programming* (*ICFP '98*), pp. 239–250. ACM Press.

Danvy, O. (1998) Functional unparsing. *J. Functional Programming*, **8**(6), 621–625.

Hinze, R. (1999) A Generic Programming Extension for Haskell. In: E. Meijer (ed), *Proc. Third Haskell Workshop*, Paris, France. (Technical Report UU-CS-1999-28, Department of Computer Science, Utrecht University.)

Jeuring, J. and Jansson, P. (1996) Polytypic Programming. In: E. Meijer and T. Sheard (eds), *Advanced Functional Programming*, pp. 68–114. *Lecture Notes in Computer Science* **1129**. Springer-Verlag.

Nordström, B., Petersson, K. and Smith, J. (1990) *Programming in Martin-Löf's Type Theory. An Introduction*, pp. 98–100 Oxford University Press.

Peyton Jones, S. and Hughes, J. (eds). (1999a) *Report on the Programming Language Haskell 98*. `http://www.haskell.org/onlinereport/`

Peyton Jones, S. and Hughes, J. (eds). (1999b) *Standard Libraries for the Haskell 98 Programming Language*. `http://www.haskell.org/onlinelibrary/`

Wadler, P. (1990) Deforestation: Transforming Programs to Eliminate Trees. *Theor. Comput. Sci.* **73**(2), 231–248.