BRICS

Basic Research in Computer Science

BRICS RS-01-10 Fridlender & Indrika: Do we Need Dependent Types?

Do we Need Dependent Types?

Daniel Fridlender
Mia Indrika

BRICS Report Series RS-01-10

ISSN 0909-0878 March 2001

See back inner page for a list of recent BRICS Report Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/01/10/`

Do we need dependent types?*

Daniel Fridlender†

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

Mia Indrika

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

Abstract

Inspired by [1], we describe a technique for defining, within the Hindley-Milner type system, some functions which seem to require a language with dependent types. We illustrate this by giving a general definition of `zipWith` for which the Haskell library provides a family of functions, each member of the family having a different type and arity. Our technique consists in introducing ad hoc codings for natural numbers which resemble numerals in λ-calculus.

*This is a summary of a paper with the same title appeared in JFP [2].

†Current affiliation: FaMAF, Universidad Nacional de Córdoba, Argentina.

‡Basic Research in Computer Science,
Centre of the Danish National Research Foundation.

1 The problem

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

Consider for example the following scheme defining `zipWith`.

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

Figure 1: Scheme for `zipWith`.

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. But, as mentioned above, Haskell does not allow dependent types.

The way the Haskell library [4, 5] 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.

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

[1]In the Haskell library, `zipWith0`, `zipWith1` and `zipWith2` are called `repeat`, `map` and `zipWith` respectively.

to “zip” lists of integers, booleans, or of values of any other type. But still their definitions are 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 [3].

2 A preliminary solution

As a motivating example, suppose we want to “zip” 8 given lists `as1`, ..., `as8` with a given 8-ary function `f` of appropriate type in Haskell. Because of 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, that is:

```
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
```

We revisit now the motivating example from Section 2.

4 The numerals in use

Assume that the numeral `seven` is defined in the library. In order to "zip" 8 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 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
```

See [2] for an example of how to reuse these numerals in some situations.

5 Conclusion

Inspired by the work presented in [1], we considered a function whose implementability was generally believed to require dependent types. We have shown that it is possible to define it without dependent types in an elegant way by introducing ad hoc numerals. The same technique can be applied to other functions like `liftM`, `zip`, `unzip`, `curry`, `uncurry`, and `taut`. The case of `taut` is explained in detail in [2].

The reader is referred to [2] for further discussions about our solution: the problem of numerals being too ad hoc, the role of polymorphism, the orthogonality with strictness and lazyness, the performance of our `zipWith`. In addition, our solution is there compared to solving the problem in languages with dependent types and in languages for generic programming.

Acknowledgments

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

[1] O. Danvy. Functional Unparsing. *Journal of Functional Programming*, 8(6):621–625, 1998.

[2] D. Fridlender and M. Indrika. Do we need dependent types? *Journal of Functional Programming*, 10(4):409–415, 2000.

[3] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

[4] S. Peyton Jones and J. Hughes, editors. *Report on the Programming Language Haskell 98.* `http://www.haskell.org/onlinereport/`, 1999.

[5] S. Peyton Jones and J. Hughes, editors. *Standard Libraries for the Haskell 98 Programming Language.* `http://www.haskell.org/onlinelibrary/`, 1999.

Recent BRICS Report Series Publications

RS-01-10 **Daniel Fridlender and Mia Indrika.** ***Do we Need Dependent Types?*** **March 2001. 6 pp. Appears in *Journal of Functional Programming*, 10(4):409–415, 2000. Superseeds BRICS Report RS-98-38.**

RS-01-9 **Claus Brabrand, Anders Møller, and Michael I. Schwartzbach.** ***Static Validation of Dynamically Generated HTML.*** **February 2001. 18 pp.**

RS-01-8 **Ulrik Frendrup and Jesper Nyholm Jensen.** ***Checking for Open Bisimilarity in the π-Calculus.*** **February 2001. 61 pp.**

RS-01-7 **Gregory Gutin, Khee Meng Koh, Eng Guan Tay, and Anders Yeo.** ***On the Number of Quasi-Kernels in Digraphs.*** **January 2001. 11 pp.**

RS-01-6 **Gregory Gutin, Anders Yeo, and Alexey Zverovich.** ***Traveling Salesman Should not be Greedy: Domination Analysis of Greedy-Type Heuristics for the TSP.*** **January 2001. 7 pp.**

RS-01-5 **Thomas S. Hune, Judi Romijn, Mariëlle Stoelinga, and Frits W. Vaandrager.** ***Linear Parametric Model Checking of Timed Automata.*** **January 2001. 44 pp. To appear in *Tools and Algorithms for The Construction and Analysis of Systems: 7th International Conference*, TACAS '01 Proceedings, LNCS, 2001.**

RS-01-4 **Gerd Behrmann, Ansgar Fehnker, Thomas S. Hune, Kim G. Larsen, Paul Pettersson, and Judi Romijn.** ***Efficient Guiding Towards Cost-Optimality in* UPPAAL.** **January 2001. 21 pp. To appear in *Tools and Algorithms for The Construction and Analysis of Systems: 7th International Conference*, TACAS '01 Proceedings, LNCS, 2001.**

RS-01-3 **Gerd Behrmann, Ansgar Fehnker, Thomas S. Hune, Kim G. Larsen, Paul Pettersson, Judi Romijn, and Frits W. Vaandrager.** ***Minimum-Cost Reachability for Priced Timed Automata.*** **January 2001. 22 pp. To appear in *Hybrid Systems: Computation and Control*, 2001.**

RS-01-2 **Rasmus Pagh and Jakob Pagter.** ***Optimal Time-Space Trade-Offs for Non-Comparison-Based Sorting.*** **January 2001. ii+20 pp.**