An n-ary `zipWith` in Haskell

Daniel Fridlender
Mia Indrika

BRICS Report Series RS-98-38

ISSN 0909-0878 December 1998

An n-ary `zipWith` in Haskell

Daniel Fridlender[*][†] Mia Indrika[‡]

Abstract

The aim of this note is to present an alternative definition of the `zipWith` family in the Haskell Library Report [5]. Because of the difficulties in defining a well-typed function with a variable number of arguments, [5] presents a family of `zipWith` functions. It provides zip functions $\texttt{zipWith}_2, \texttt{zipWith}_3, \ldots, \texttt{zipWith}_7$. For each n, $\texttt{zipWith}_n$ zips n lists with a n-ary function. Defining a single `zipWith` function with a variable number of arguments seems to require dependent types. Inspired by [3], we show, however, how to define such a function in Haskell by means of a binary operator for grouping its arguments. For comparison, we also give definitions of `zipWith` in languages with dependent types.

1 `zipWith` in the Haskell library

The Haskell library [5] supplies the programmer with a family of functions $\texttt{zipWith}_2$, $\texttt{zipWith}_3$, …, $\texttt{zipWith}_7$ indexed by the number of lists to which it applies.

For instance, $\texttt{zipWith}_4$ is defined as follows.

[*]BRICS, Basic Research in Computer Science,
Centre of the Danish National Research Foundation.
[†]Department of Computer Science, University of Aarhus,
Building 540, Ny Munkegade, DK-8000 Århus C, Denmark.
E-mail: `daniel@brics.dk`
[‡]Department of Computing Science,
Chalmers University of Technology and Göteborg University,
412 96 Göteborg, Sweden.
E-mail: `indrika@cs.chalmers.se`

```
zipWith4 :: (a->b->c->d->e) -> [a]->[b]->[c]->[d]->[e]

zipWith4 z (a:as) (b:bs) (c:cs) (d:ds)
                    = z a b c d : zipWith4 z as bs cs ds
zipWith4 _ _ _ _ _ = []
```

In addition `zipWith`$_2$ is simply called `zipWith` and the function `map` can actually be thought of as the function `zipWith`$_1$.

The programmer rarely needs a `zipWith`$_n$ with $n > 7$, the idea is that if it is necessary, he or she can follow the same pattern of those in the library to implement it.

We want to define a well-typed function `zipWith` which given a function `f` of type `(a1->a2->...->an->b)` and n lists of types `[a1]`, `[a2]`, ... `[an]`, respectively, returns a list of type `[b]` obtained by zipping the n lists with the function `f`.

Defining such a general version of `zipWith` seems to require dependent types. Section 4 shows examples of how to write that function in Agda [2], a functional language with dependent types. However, we show in Section 2 that it is also possible to define a general `zipWith` in Haskell, a language which does not have dependent types.

We will compare the different alternatives from the point of view of a user of `zipWith` concerning what has to be written in order to use `zipWith`. For that comparison we will show how to write an expression that zips the lists `[1..]`, `"hi"` and `"world"`, with the ternary function `(,,)` which constructs a triple from its arguments.

Using the definitions from the Haskell library, we would write

```
zipWith3 (,,) [1..] "hi" "world"
```

which gives as result

```
[(1, 'h', 'w'), (2, 'i', 'o')].
```

2 An n-ary `zipWith` function

As mentioned above, defining a general version of `zipWith` seems to require dependent types. However, as already shown by Danvy [3] for the `printf` function, it is sometimes possible to represent such functions in a language without dependent types. We apply a similar idea to the example of `zipWith`.

2.1 A first solution

A first solution essentially consists of replacing each of the lists with a list-of-functions transformer by means of the function `inzip`.

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

Given a list `as1` of type `[a1]`, `inzip as1` transforms lists whose elements are functions of type `a1->b` (for any type `b`) into lists of type `[b]`. Given in addition a list `as2` of type `[a2]`, the composition of the transformer `inzip as2` with `inzip as1` gives another transformer, which now transforms lists of functions of type `a1->a2->b` (for any type `b`) into lists of type `[b]`.

```
inzip as2 . inzip as1 :: [a1->a2->b] -> [b]
```

This leads us to define the following composition between transformers:

```
(~~~) :: (a->b) -> (b->c) -> (a->c)
(~~~) = flip (.)
```

Hence, given n lists `as1`, `as2`, ..., `asn` of types `[a1]`, `[a2]`, ..., `[an]`, respectively, by composition one obtains the following transformer:

```
inzip as1 ~~~ inzip as2 ~~~ ... ~~~ inzip asn
      :: [a1->a2->...->an->b] -> [b]
```

It only remains to create a list of functions on which this transformer will operate. Given a function `f` of type `a1->a2->...->an->b`, that list will consist of the infinite repetition of `f`.

Thus, given a function and a transformer the function `zipWith` is implemented as follows.

```
zipWith f t = t (repeat f)
```

In general, the function `f` will be of type `a1->a2->...->an->b` and the transformer `t` of type `[a1->a2->...->an->b] -> [b]`. This means that we want `zipWith` to have type

```
(a1->a2->...->an->b) -> ([a1->a2->...->an->b]->[b]) -> [b]
```

for every n. Again, the type of `zipWith` depends on the number of lists to be zipped. But fortunately, `zipWith` as defined above admits a more general and simple type.

```
zipWith :: a -> ([a]->b) -> b
```

With this, our example would be written

```
zipWith (,,) (inzip [1..] ~~~ inzip "hi" ~~~ inzip "world").
```

2.2 Another solution

Still, we would like to be able to use `zipWith` without having to type in so long transformers. One way to do that, is to define an operator `~~` which combines `inzip` with the operator `~~~`. It should satisfy the equation.

```
as ~~ rest = inzip as ~~~ rest
```

This equation, which can be taken as its definition, tells us that `~~` associates to the right and that its type is

```
(~~) :: [a] -> ([b] -> c) -> [a->b] -> c
```

A way to understand `~~` is that it applies each function from its third argument to the corresponding element in the first argument giving an intermediate result of type `[b]`. To this, the second argument is applied to obtain the final result. The second argument is thus a continuation.

The example given above can be written now

```
zipWith (,,) ([1..] ~~ "hi" ~~ inzip "world")
```

or also, using the initial continuation `id` and the law `as ~~ id = inzip as`,

```
zipWith (,,) ([1..] ~~ "hi" ~~ "world" ~~ id)
```

Disregarding the unpleasant presence of `id`, we can think of the operator `~~` as a way of grouping the lists to be zipped.

3 Performance

From the point of view of the computation, using the solution above cannot be more efficient than using the one in [5]. Whether or not it is (strictly) less efficient depends on the techniques for deforestation [6] that the compiler has. This is because the extra computation of our solution occurs when creating and consuming the infinite list produced with the function `repeat`. With advanced deforestation techniques the compiler will transform the applications of `zipWith` into applications of functions equivalent to those provided in the Haskell library.

Even without advanced techniques, the loss in performance is not significant.

4 `zipWith` with dependent types

As mentioned before, a possibility for well typing a general `zipWith` is by means of dependent types like in Agda [2] or Cayenne [1].

In such languages a natural way to write `zipWith` is by defining a type which depends on the arity. This is the main idea behind the solution presented in the Appendix A for the language Agda. Observe that, because of the limitations of type inference in the presence of dependent types, much more type information must be explicitly given in the program. For that reason, besides the arity `n`, a family of types `A` indexed by positive numbers is an argument of `zipWith`. In practice, only a finite initial segment of the family is important, namely `A 1`, `A 2`, . . . , `A n`.

With this solution, the example above would be written as follows (using Haskell-like notations to harmonize with the previous example)

```
A (n :: Pos) :: Set =
  case n of
    (one)  -> Int
    (s n') -> case n' of
                (one)   -> Char
                (s n'') -> Char

zipWithN A 3 (Int,Char,Char) (,,) [1..] "hi" "world"
```

Appendix B gives another definition in Agda as proposed by Lennart Augustsson. In this solution, the type information is given as a list of types. The length of the list provides implicitly the arity. Thus, instead of using the family of types `A` and arity `n` the list would consist of the types `A 1`, `A 2`, . . . , `A n`. This is possible thanks to the ability of the language to define a type (actually, a kind) `ListT` whose elements are lists of types.

Using this solution, one can write

```
argsT :: ListT
argsT = ConsT Int (ConsT Char (OneT Char))

zipWithT argsT (Int,Char,Char) (,,) [1..] "hi" "world"
```

5 Discussion

As shown above, dependent types make possible to define elegant versions of `zipWith`. However, they have the inconvenience of forcing the user to write down type information for every application of `zipWith`. The solution given in Section 2 presents some advantage in this sense. On the other hand, having to write the initial continuation `id` is inconvenient.

We do not see a completely satisfactory way out of this problem. An alternative would be to have an extra operator `~~~~` to be used for grouping the last two lists, writing for example

```
zipWith f (as ~~ bs ~~ cs ~~~~ ds)
```

instead of

```
zipWith f (as ~~ bs ~~ cs ~~ ds ~~ id).
```

Another alternative could be to define `begin` and `end` so that one can write instead

```
zipWith f (begin ~~ as ~~ bs ~~ cs ~~ ds ~~ end).
```

In this last case, the definition of `zipWith` needs to be slightly modified.

Notice that it is impossible to type in Haskell a general function `zipWith` such that we can just write

```
zipWith f as1 as2 ... asn
```

for every n, every `f` of type `a1->a2->...->an->b`, and every `as1`, `as2`, ..., `asn` of type `[a1]`, `[a2]`, ..., `[an]`, respectively. To see this, consider the case $n = 3$. The expression `zipWith f as1 as2` must be both a function and a list. A function, since it is applied to `as3`. And a list, because `f` is of type `a1->a2->x` for some type `x`, hence `zipWith f as1 as2` is a list of type `[x]`.

We remark that our definition of `zipWith` can be transformed into another definition that would work for call-by-value languages. In this case, infinite lists are implemented as streams. A stream either is empty or can be represented as a pair of its first element and a thunk (function of no arguments) that returns the rest of the stream. The initial continuation `id` becomes a function that converts a stream into a list. Appendix C presents a definition of `zipWith` in the language Scheme [4].

As Magnus Carlsson pointed out to us, from the point of view of a Haskell programmer it may be convenient to write

```
repeat (,,) `zap` [1..] `zap` "hi" `zap` "world"
```

for our example, where `zap` associates to the left and can be defined as `flip inzip`. This captures the computation behind the definition of `zipWith` and `~~` in Section 2 in a more natural way. But this avoids one of the issues we wanted to address here: the one of defining without dependent types a function (`zipWith`) which seems to require them.

He also proposed to apply similar techniques to other families of functions in the Haskell library, like the family of `liftM`.

6 Acknowledgments

We are grateful to Magnus Carlsson and Olivier Danvy for their interest and useful discussions.

References

[1] Lennart Augustsson. Cayenne — a language with dependent types. `http://www.md.chalmers.se/~augustss/cayenne/paper.ps`, 1998.

[2] Catarina Coquand. Agda documentation. `http://www.md.chalmers.se/~catarina/agda/doc.html`, 1998.

[3] Olivier Danvy. Functional unparsing. Technical Report BRICS RS-98-12, Department of Computer Science, University of Aarhus, Aarhus, Denmark, May 1998. Supersedes the earlier report BRICS RS-98-5. Extended version of an article to appear in the Journal of Functional Programming.

[4] R. Kelsey, W. Clinger, and J. Rees (editors). Revised^5 Report on the Algorithmic Language Scheme. Report, February 1998.

[5] J. Peterson, K. Hammond, L. Augustsson, J. Fasel, A. Gordon, M. Jones, S. Peyton Jones, and A. Reid. The Haskell Library Report Version 1.4. `http://haskell.org/onlinelibrary/`, April 1997.

[6] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990. (Special issue of selected papers from 2'nd ESOP.).

A First `zipWith` in Agda

```
data Pos = one | s (n :: Pos)
data List (A :: Set) = nil | cons (a :: A) (as :: List A)
data Pair (A, B :: Set) = pair (a :: A) (b :: B)

(^) (A :: Pos -> Set) (n :: Pos) :: Set =
  case n of
    (one)  -> A one@_
    (s n') -> Pair (A^n') (A n)
```

```
Fun (A :: Pos -> Set) (n :: Pos) (B :: Set) :: Set =
  case n of
    (one)  -> A one@_ -> B
    (s n') -> Fun A n' (A n -> B)

fst (A,B :: Set) (p :: Pair A B) :: A =
  case p of (pair a b) -> a
snd (A,B :: Set) (p :: Pair A B) :: B =
  case p of (pair a b) -> b

(*) (A :: Pos -> Set) (n :: Pos) :: Set = List (A n)

zip (A,B :: Set) (as :: List A) (bs :: List B) :: List (Pair A B) =
  case as of
    (nil)        -> nil@_
    (cons a as') ->
      case bs of
        (nil)        -> nil@_
        (cons b bs') -> cons@_ (pair@_ a b) (zip A B as' bs')

map (A,B :: Set) (f :: A -> B) (as :: List A) :: List B =
  case as of
    (nil)        -> nil@_
    (cons a as') -> cons@_ (f a) (map A B f as')

currys (A :: Pos -> Set) (n :: Pos) (B :: Set) (f :: A^n -> B)
  :: Fun A n B =
  case n of
    (one)  -> f
    (s n') -> currys A
                     n'
                     (A n -> B)
                     (\(a :: A^n') -> \(b :: A n) -> f (pair@_ a b))

uncurrys (A :: Pos -> Set) (n :: Pos) (B :: Set) (f :: Fun A n B)
  :: A^n -> B =
  case n of
    (one)  -> f
    (s n') -> \(ab :: Pair (A^n') (A n)) ->
                uncurrys A
                         n'
                         (A n -> B)
                         f
                         (fst (A^n') (A n) ab)
                         (snd (A^n') (A n) ab)
```

```
zipTN (A :: Pos -> Set) (n :: Pos) (ts :: ((*) A)^n) :: List (A^n) =
  case n of
    (one)  -> ts
    (s n') -> case ts of
                (pair ts' as) ->
                  zip (A^n') (A n) (zipTN A n' ts') as

zipN (A :: Pos -> Set) (n :: Pos) :: Fun ((*) A) n (List (A^n)) =
  currys ((*) A) n (List (A^n)) (zipTN A n)

zipWithTN (A :: Pos -> Set) (n :: Pos) (B :: Set) (f :: A^n -> B)
  (ts :: ((*) A)^n) :: List B =
  map (A^n) B f (zipTN A n ts)

zipWithN (A :: Pos -> Set) (n :: Pos) (B :: Set) (f :: Fun A n B)
  :: Fun ((*) A) n (List B) =
  currys ((*) A) n (List B) (zipWithTN A n B (uncurrys A n B f))
```

B Second zipWith in Agda

```
ListT :: Type = data oneT (a :: Set) | consT (a :: Set) (ts :: ListT)
oneT (a :: Set) :: ListT = oneT@_ a
consT (a :: Set) (ts :: ListT) :: ListT = consT@_ a ts

LListT (ts :: ListT) :: ListT =
  case ts of
    (oneT t)     -> oneT (List t)
    (consT t ts) -> consT (List t) (LListT ts)

FunT (ts :: ListT) (a :: Set) :: Set =
  case ts of
    (oneT t)     -> t -> a
    (consT t ts) -> t -> FunT ts a

ProdT (ts :: ListT) :: Set =
  case ts of
    (oneT t)     -> t
    (consT t ts) -> Pair t (ProdT ts)

curryT (ts :: ListT) (a :: Set) (f :: ProdT ts -> a) :: FunT ts a =
  case ts of
    (oneT t)     -> \(x :: t) -> f x
    (consT t ts) -> \(x :: t) ->
                      curryT ts a (\(xs :: ProdT ts) -> f (pair@_ x xs))
```

```
uncurryT (ts :: ListT) (a :: Set) (f :: FunT ts a) :: ProdT ts -> a =
  case ts of
    (oneT t)     ->  \(x :: t) -> f x
    (consT t ts) ->  \(p :: ProdT (consT t ts)) ->
                       case p of
                         (pair x xs) -> uncurryT ts a (f x) xs

zipT (ts :: ListT) (xs :: ProdT (LListT ts)) :: List (ProdT ts) =
  case ts of
    (oneT t)     -> xs
    (consT t ts) -> zip t
                        (ProdT ts)
                        (fst (List t) (ProdT (LListT ts)) xs)
                        (zipT ts (snd (List t) (ProdT (LListT ts)) xs))

zipWithT (ts :: ListT) (a :: Set) (f :: FunT ts a)
  :: FunT (LListT ts) (List a) =
  curryT (LListT ts)
         (List a)
         (\(xs :: ProdT (LListT ts)) -> map (ProdT ts)
                                             a
                                            (uncurryT ts a f)
                                            (zipT ts xs)
         )
```

C zipWith in Scheme

```
(define make-stream
  (lambda (value thunk) (cons value thunk)))

(define null-stream '())

(define stream-null?
  (lambda (s) (eq? s null-stream)))

(define tail
  (lambda (s) ((cdr s))))

(define head car)

(define repeat
  (lambda (v) (make-stream v (lambda () (repeat v)))))
```

```
(define inzip
  (lambda (l)
    (lambda (s) (if (or (null? l) (stream-null? s))
                    null-stream
                    (make-stream
                      ((head s) (car l))
                      (lambda () ((inzip (cdr l)) (tail s))))))))

(define ~~~
  (lambda (t1 t2)
    (lambda (k) (t2 (t1 k)))))

(define ~~
  (lambda (l t) (~~~ (inzip l) t)))

(define id
  (lambda (s) (if (stream-null? s)
                  '()
                  (cons (head s) (id (tail s))))))

(define zipWith
  (lambda (f t) (t (repeat f))))
```