PolyLib - a polytypic function library

PolyLib — a library of polytypic functions

Patrik Jansson Johan Jeuring

May 15, 1998

1 Introduction

A polytypic program is a program that behaves uniformly over a large class of datatypes. For functional polytypic programming this uniformity is achieved by parameterising functions over type constructors to obtain *polytypic functions* [12]. A polytypic function is defined either by induction on the structure of *regular*[1] type constructors or in terms of other polytypic functions. PolyP [8] is an extension of the functional programming language Haskell with a construct for defining polytypic functions. PolyP is a type guided preprocessor that generates instances of polytypic functions and inserts applications of these instances where needed.

During the last few years we have used PolyP to construct a number of polytypic programs, for example for unification, parsing, rewriting, pattern matching, etc. These polytypic programs use several basic polytypic functions, such as the relatively well-known `cata` and `map`, but also less well-known functions such as `propagate` and `thread`. We have collected these basic polytypic functions in the library of PolyP: PolyLib [10, app. B]. This paper describes the polytypic functions in PolyLib, motivates their presence in the library, and gives a rationale for their design. Thus we hope to share our experience with other researchers in the field. We will assume the reader has some familiarity with the field of polytypic programming.

Of course, a library is an important part of a programming language. Languages like Java, Delphi, Perl and Haskell are popular partly because of their useful and extensive libraries. For a polytypic programming language it is even more important to have a clear and well-designed library: writing polytypic programs is difficult, and we do not expect many programmers to *write* polytypic programs. On the other hand, many programmers *use* polytypic programs such as parser generators, equality functions, etc.

This is a first attempt to describe the library of PolyP; we expect that both the form and content of this description will change over time. One of the goals of this paper is to obtain feedback on the library design from other researchers working within the field. At the moment the library only contains the basic

[1]A type constructor `d` is regular if the datatype `d a` contains no function spaces, and if the argument of `d` is the same on the left- and right-hand side of its definition.

polytypic functions. In the future we will develop special purpose sub-libraries for polytypic functions with more advanced functionality, for example for parsing and the other programs mentioned above.

2 Polytypic programs

Using different versions of PolyP (and its predecessors) we have implemented a number of polytypic programs. For example, we have implemented a polytypic equality function, a polytypic show function, and a polytypic parser. Furthermore, we have implemented some more involved polytypic programs for pattern matching, unification and rewriting. Brief descriptions of these programs are given in section 4. This section introduces the format we use for describing polytypic library functions, and gives an overview of the contents of the library.

2.1 Describing polytypic functions

Our description of a polytypic function consists of (some of) the following components: its name and type; an (in)formal description of the function; other names the function is known by; known uses of the function; and its background and relationship to other polytypic functions. (Here we will often refer to the polytypic applications in section 4.) For example:

```
pmap :: (a -> b) -> d a -> d b
```

Function `pmap` takes a function `f` and a value `x` of datatype `d a`, and applies `f` ... **Also known as**: `map` [12], `map`$_n$ [11]. **Known uses**: Everywhere! **Background**: This was one of the first ...

A problem with describing a library of polytypic functions is that it is not completely clear how to *specify* polytypic functions. The most basic combinators have immediate category theoretic interpretations that can be used as a specification, but for more complicated combinators the matter is not all that obvious. Thus, we will normally not provide formal specifications of the library functions, though we try to give references to more in-depth treatments.

2.2 Library overview

We have divided the library in five parts, see figure 1. The first part of the library contains powerful recursion combinators such as `map`, `cata` and `ana`. This part is the core of the library in the sense that it is used in the definitions of all the functions in the other parts. The second part deals with zips and some derivates, such as the equality function. The third part consists of functions that manipulate monads. The fourth and fifth parts consist of simpler (but still very useful) functions, like flattening and summing. The following section describes each of these functions in more detail.

`pmap`, `fmap`, `cata`
`ana`, `hylo`, `para`
`crush`, `fcrush`

(a) Recursion operators

`pzip`, `fzip`
`punzip`, `funzip`
`pzipWith`, `pzipWith'`
`pequal`, `fequal`

(b) Zips etc.

`pmapM`, `fmapM`, `cataM`
`anaM`, `hyloM`, `paraM`
`propagate`, `cross`
`thread`, `fthread`

(c) Monad op's

`flatten`, `fflatten`
`fl_par`, `fl_rec`, `conc`

(d) Flatten functions

`psum`, `size`, `prod`
`pand`, `pall`
`por`, `pany`, `pelem`

(e) Miscellaneous

Figure 1: Overview of PolyLib

3 PolyLib

For the polytypic functions that have Haskell counterparts we prepend the letter `p` (for polytypic) to the Haskell name to avoid a name clash. (The bifunctor variants instead begin with an `f`.) In types we sometimes use `b <- a` as syntactic sugar for `a -> b`. A polytypic function can be thought of as taking (a representation of) a functor as its first argument. This argument is normally omitted but sometimes written as a subscript for clarity: $\texttt{pmap}_d$.

The polytypic functions below are only defined for regular datatypes `d a`. In the type this is indicated by adding a context `Regular d => ...`, but we will omit this for brevity. A regular type constructor `d` is always the fixpoint of some bifunctor `f`. We will denote this `f` by `FunctorOf d`.

3.1 Recursion operators

```
pmap :: (a -> b) -> d a -> d b
fmap :: (a -> c) -> (b -> d) -> f a b -> f c d
```

Function `pmap` takes a function `f` and a value `x` of datatype `d a`, and applies `f` recursively to all occurrences of elements of type `a` in `x`. With `d` as a functor acting on types, $\texttt{pmap}_d$ is the corresponding functor action on functions. Function $\texttt{fmap}_f$ is the corresponding functor action for a bifunctor `f`. **Also known as**: `map` [12], $\texttt{map}_n$ [11]. In **charity** [3] $\texttt{map}_d$ `f x` is written `d f (x)`. **Known uses**: Everywhere! Function `fmap` is used in the definition of `pmap`, `cata`, `ana`, `hylo`, `para` and in many other PolyLib functions. **Background**: The map function was one of the first combinators distinguished in the work of Bird and Meertens, [2, 16]. The traditional `map` in functional languages maps a function over a list of elements. The current Haskell version of map is overloaded:

```
map :: Functor f => (a->b) -> f a -> f b
```

and can be used as the polytypic `pmap` if instance declarations for all regular type constructors are given. Function `pmap` can be used to give default instances for the Haskell `map`.

```
cata :: (FunctorOf d a b -> b) -> (d a -> b)
ana  :: (FunctorOf d a b <- b) -> (d a <- b)
hylo :: (f a b -> d) -> (c -> f a b) -> (c -> d)
para :: (d a -> FunctorOf d a b -> b) -> (d a -> b)
```

Four powerful recursion operators on the type `d a`: The catamorphism, `cata`, "evaluates" a data structure by recursively replacing the constructors with functions. The typing of `cata` may seem unfamiliar but with the definition of `FunctorOf` above it can seen as equivalent to:

```
cata :: (f a b -> b) -> (Mu f a -> b)
```

The anamorphism, `ana`, works in the opposite direction and builds a data structure. The hylomorphism, `hylo`, is the generalisation of these two functions that simultaneously builds and evaluates a structure. Finally, the paramorphism, `para`, is a generalised form of `cata` that gives its parameter function access not only to the results of evaluating the substructures, but also the structure itself. **Also known as**:

PolyLib	Functorial ML [1]	Squiggol	**charity** [3]
`cata i`	$\texttt{fold}_1$ `i`	$(\!|i|\!)$	`i`
`ana o`	-	$[\!(o)\!]$	`( o )`

Functions `cata` and `para` are instances of the Visitor pattern in [5]. **Known uses**: Very many polytypic functions are defined using `cata`: `pmap`, `crush`, `thread`, `flatten`, `propagate`, and all our applications use it. Function `para` is used in `rewrite`. **Background**: The catamorphism, `cata`, is the generalisation of the Haskell function `foldr` and the anamorphism, `ana`, is the (category theoretic) dual. Catamorphisms were introduced by Malcolm [14, 15]. A hylomorphism is the fused composition of a catamorphism and an anamorphism specified by: `hylo i o = cata i . ana o`. The paramorphism [17], `para`, is the elimination rule for the type `d a` from type theory. It captures the recursion pattern of primitive recursive functions on the datatype `d a`.

```
crush  :: (a->a->a) -> a -> d a -> a
fcrush :: (a->a->a) -> a -> f a a -> a
```

The function `crush op e` takes a structure `x` and inserts the operator `op` from left to right between every pair of values of type `a` at every level in `x`. (The value `e` is used in empty leaves.) **Known uses**: within the library see section 3.5. Many of the functions in that section are then used in the different applications. **Background**: The definition of `crush` is found in [18]. For an associative operator `op` with unit `e`, `crush op e` can be defined as `foldr op e . flatten`. As `crush` has the same arguments as `fold` on lists it can be seen as an alternative to `cata` as the generalisation of `fold` to regular datatypes.

3.2 Zips

```
pzip   :: (d a,d b) -> Maybe ( d (a,b) )
punzip :: d (a,b) -> (d a,d b)
fzip   :: (f a b,f c d) -> Maybe ( f (a,c) (b,d) )
funzip :: f (a,c) (b,d) -> (f a b,f c d)
```

Function `punzip` takes a structure containing pairs and splits it up into a pair of structures containing the first and the second components respectively. Function `pzip` is a partial inverse of `punzip`: it takes a pair of structures and zips them together to `Just` a structure of pairs if the two structures have the same shape, and to `Nothing` otherwise. **Also known as**: $\texttt{zip}_m$ [11], **zip.×.d** [6], **Known uses**: Function `fzip` is used in the definition of `pzipWith`. **Background**: The traditional function `zip`

```
zip :: [a] -> [b] -> [(a,b)]
```

combines two lists and does not need the `Maybe` type in the result as the longer list can always be truncated. (In general such truncation is possible for all types that have a nullary constructor, but not for all regular types.) A more general ("doubly polytypic") variant of `pzip`: `transpose` (called **zip.d.e** in [6])

```
transpose :: d (e a) -> e (d a)
```

was first described by Fritz Ruehr [19]. For a formal definition, see Hoogendijk & Backhouse [6].

```
pzipWith  :: ((a,b) -> Maybe c) -> (d a,d b) -> Maybe (d (a,b))
pzipWith' :: (FunctorOf d c e -> e) -> ((d a, d b) -> e) ->
             ((a, b) -> c) -> (d a,d b) -> e
```

Function `pzipWith op` works like `pzip` but uses the operator `op` to combine the values from the two structures instead of just pairing them. As the zip might fail, we also give the operator a chance to signal failure by giving it a `Maybe`-type as a result.[2]

[2]The type constructor `Maybe` can be replaced by any monad with a zero, but we didn't want to clutter up the already complicated type with contexts.

Function `pzipWith'` is a generalisation of `pzipWith` that can handle two structures of different shape. In the call `pzipWith' ins fail op`, `op` is used as long as the structures have the same shape, `fail` is used to handle the case when the two structures mismatch, and `ins` combines the results from the substructures. (The type of `ins` is the same as the type of the first argument to `cata`.) **Also known as**: $\texttt{zipop}_m$ [11]. **Known uses**: Function `pzipWith'` is used in the definition of equality, matching and even unification. **Background**: Function `pzipWith` is the polytypic variant of the Haskell function `zipWith`

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

but `pzipWith'` is new. Function `pzip` is just `pzipWith Just`.

```
pequal :: (a->b->Bool) -> d a -> d b -> Bool
fequal :: (a->b->Bool) -> (c->d->Bool) -> f a c -> f b d -> Bool
```

The expression `pequal eq x y` checks if the structures `x` and `y` are equivalent using the equivalence operator `eq` to compare the elements pairwise. **Known uses**: `fequal` is used in the unification algorithm to determine when two terms are top level equal. **Background**: An early version of a polytypic equality function appeared in [20]. Function `pequal` can be instantiated to give a default for the Haskell `Eq`-class for regular datatypes:

```
(==) :: Eq a => d a -> d a -> Bool
(==) =  pequal (==)
```

In Haskell the equality function can be automatically derived by the compiler, and our polytypic equality is an attempt at moving that derivation out of the compiler into the prelude.

3.3 Monad operations

```
pmapM :: Monad m => (a->m b) -> d a -> m (d b)
fmapM :: Monad m => (a->m c) -> (b->m d) -> f a b -> m (f c d)
cataM :: Monad m => (FunctorOf d a b -> m b) -> (d a -> m b)
anaM  :: Monad m => (b -> m (FunctorOf d a b)) -> (b -> m (d a))
hyloM :: Monad m => (f a b -> m d) -> (c -> m (f a b)) -> c -> m d
paraM :: Monad m => (d a -> FunctorOf d a b -> m b) -> d a -> m b
```

Function `pmapM` is a variant of `pmap` that threads a monad `m` from left to right through a structure after applying its function argument to all elements in the structure. A monadic map can, for example, use a state monad to record information about the elements in the structure during the traversal. The other recursion operators are generalised in the same way to form even more general combinators. **Also known as**: traversals [11]. **Known uses**: in `unify` and in the parser. **Background**: Monadic maps and catamorphisms are described in [4]. The monadic map (also called active traversal) is closely related to `thread` (also called passive traversal):

```
pmapM f   =   thread . pmap f
thread    =   pmapM id
```

```
propagate :: d (Maybe a) -> Maybe (d a)
cross     :: d [a] -> [d a]
```

Function `propagate` propagates `Nothing` to the top level. Function `cross` is the cross (or tensor) product that given a structure `x` containing lists, generates a list of structures of the same shape. This list has one element for every combination of values drawn from the lists in `x`. These two functions can be generalised to `thread` any monad through a value. **Known uses**: `propagate` is used in the definition of `pzip`. **Background**: Function `propagate` is an instance of `transpose` [19], and both `propagate` and `cross` are instances of `thread` below.

```
thread  :: Monad m => d (m a) -> m (d a)
fthread :: Monad m => f (m a) (m b) -> m (f a b)
```

Function `thread` is used to tie together the monad computations in the elements from left to right. **Also known as**: $\texttt{dist}_d$ [4]. **Known uses**: Function `thread` can be used to define the monadic map: `pmapM f = thread . pmap f`. Function `fthread` is also used in the parser to thread the parsing monad through different structures. Function `thread` can be instantiated (with `d` = `[]`) to the Haskell prelude function

```
accumulate :: Monad m => [m a] -> m [a]
```

but also (with `m` = `Maybe`) to `propagate` and (with `m` = `[]`) to `cross`.

3.4 Flatten functions

```
flatten  :: d a -> [a]
fflatten :: f a [a] -> [a]
fl_par   :: f a b -> [a]
fl_rec   :: f a b -> [b]
```

Function `flatten x` traverses the structure `x` and collects all elements from left to right in a list. The other three function are variants of this for a bifunctor

f. **Also known as**: extract$_{m,i}$ [11], **listify** [6]. **Known uses**: `fl_rec` is used in the unification algorithm to find the list of immediate subterms of a term. Function `fflatten` is used to define `flatten`

```
flatten = cata fflatten
```

Background: In the relational theory of polytypism [6] there is a membership relation **mem.d** for every relator (type constructor) **d** such that a **mem.d** x ≡ a ‘elem‘ (flatten x)

3.5 Miscellaneous

A number of simple polytypic functions can be defined in terms of `crush` and `pmap`. For brevity we present this part of PolyLib below by providing only the name, the type and the definition of each function.

```
psum :: d Int -> Int                   psum = crush (+) 0
prod :: d Int -> Int                   prod = crush (*) 1
conc :: d [a] -> [a]                   conc = crush (++) []
pand :: d Bool -> Bool                 pand = crush (&&) True
por  :: d Bool -> Bool                 por  = crush (  ) False

size    :: d a -> Int                  size    = psum . pmap (\_->1)
flatten :: d a -> [a]                  flatten = conc . pmap (:[])
pall    :: (a->Bool) -> d a -> Bool    pall p  = pand . pmap p
pany    :: (a->Bool) -> d a -> Bool    pany p  = por  . pmap p
pelem   :: Eq a => a -> d a -> Bool    pelem x = pany (\y->x==y)
```

4 Polytypic applications using PolyLib

This section lists some polytypic applications we have written during the last few years. Most of these are candidates for inclusion in future versions of PolyLib.

- A polytypic show function and a simple polytypic parser [7].
- Pattern matching, [12], can be defined for all regular datatypes that include an anonymous “wild-card”. We use a Haskell class `HasWildcard` with a member function `isWild :: t -> Bool` to express this restriction. Function `pmatch`

  ```
  pmatch :: HasWildcard t => t -> t -> Bool
  ```

  takes a pattern and a value and tries to match this pattern with the value.
- Generalising pattern matching such that it becomes symmetric in its two arguments and allows named wild-cards gives unification.

  The unification program [9]: `unify`

```
unify :: (Term t,Subst s) => t -> t -> Maybe (s t)
```

takes two terms and gives `Just` a unifying substitution if they are unifiable and `Nothing` otherwise. The member functions of the class `Term` can all be generated polytypically for every regular datatype so `unify` is also polytypic.

- Given a unification function, rewriting is not far away: in [13] we give an implementation of a function `rewrite`

  ```
  rewrite :: Term t => [(t,t)] -> t -> t
  ```

  that takes a list of rewrite rules (pairs of terms containing variables) and a term and rewrites the term as far as possible using these rules.

- A different application area is that of genetic algorithms [21], where polytypic functions for doing genetic recombinations of elements of regular datatypes are used. The recombination algorithm uses polytypic functions for extracting or replacing a certain substructure.

5 Conclusions

We have given a description of PolyLib: the library of PolyP. This library has grown out of our experience with implementing polytypic functions. PolyLib is very likely incomplete, but we think we have included most basic polytypic combinators. Future work consists of the construction of special purpose sublibraries, and of more complete description of the basic polytypic combinators. For an implementation PolyLib and PolyP, see

```
http://www.cs.chalmers.se/~patrikj/poly/polyp/
```

References

[1] G. Bellè, C.B. Jay, and E. Moggi. Functorial ML. In *PLILP'96*, volume 1140 of *LNCS*. Springer-Verlag, 1996.

[2] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume F36 of *NATO ASI Series*, pages 5–42. Springer–Verlag, 1987.

[3] Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Dep. of Computer Science, Univ. of Calgary, 1992.

[4] M.M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Memoranda Informatica 94-28, University of Twente, June 1994.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[6] Paul Hoogendijk and Roland Backhouse. When do datatypes commute? In *Category Theory and Computer Science*, LNCS 1290, pages 242–260, 1997.

[7] Marieke Huisman. The calculation of a polytypic parser. Master's thesis, Utrecht University, 1996. INF/SRC-96-19.

[8] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL'97*, pages 470–482. ACM Press, 1997.

[9] P. Jansson and J. Jeuring. Polytypic unification. *JFP*, 1998. In press.

[10] Patrik Jansson. Functional polytypic programming — use and implementation. Technical report, Chalmers Univ. of Tech., Sweden, 1997. Lic. thesis. Available from `http://www.cs.chalmers.se/~patrikj/lic/`.

[11] C.B. Jay, G. Bellè, and E. Moggi. Functorial ML. Extended version of [1] in press for JFP'98, 1998.

[12] J. Jeuring. Polytypic pattern matching. In *FPCA'95*, pages 238–248. ACM Press, 1995.

[13] J. Jeuring and P. Jansson. Polytypic programming. In *AFP'96*, volume 1129 of *LNCS*, pages 68–114. Springer-Verlag, 1996.

[14] G. Malcolm. Homomorphisms and promotability. In J.L.A. van de Snepscheut, editor, *Mathematics of Program Construction*, pages 335–347. Springer-Verlag, 1989. LNCS 375.

[15] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

[16] L. Meertens. Algorithmics — towards programming as a mathematical activity. In *Proceedings of the CWI Symposium on Mathematics and Computer Science*, volume 1 of *CWI Monographs*, pages 289–334. North–Holland, 1986.

[17] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–425, 1992.

[18] L. Meertens. Calculate polytypically! In *PLILP'96*, volume 1140 of *LNCS*, pages 1–16. Springer Verlag, 1996.

[19] Fritz Ruehr. *Analytical and Structural Polymorphism Expressed Using Patterns Over Types.* PhD thesis, University of Michigan, 1992.

[20] Tim Sheard. Automatic generation and use of abstract structure operators. *ACM TOPLAS*, 13(4):531–557, 1991.

[21] Måns Vestin. Genetic algorithms in Haskell with polytypic programming. Master's thesis, Göteborg University, Gothenburg, Sweden, 1997. Available from `http://www.cs.chalmers.se/~johanj/polytypism/genetic.ps`.