PolyP — a polytypic programming language extension

Patrik Jansson and Johan Jeuring
Chalmers University of Technology and University of Göteborg
S-412 96 Göteborg, Sweden
email: {`patrikj,johanj`}`@cs.chalmers.se`
url: `http://www.cs.chalmers.se/~{patrikj,johanj}/`

Abstract

Many functions have to be written over and over again for different datatypes, either because datatypes change during the development of programs, or because functions with similar functionality are needed on different datatypes. Examples of such functions are pretty printers, debuggers, equality functions, unifiers, pattern matchers, rewriting functions, etc. Such functions are called polytypic functions. A polytypic function is a function that is defined by induction on the structure of user-defined datatypes. This paper extends a functional language (a subset of Haskell) with a construct for writing polytypic functions. The extended language type checks definitions of polytypic functions, and infers the types of all other expressions using an extension of Jones' theories of qualified types and higher-order polymorphism. The semantics of the programs in the extended language is obtained by adding type arguments to functions in a dictionary passing style. Programs in the extended language are translated to Haskell.

1 Introduction

Complex software systems usually contain many datatypes, which during the development of the system change regularly. Developing innovative and complex software is typically an evolutionary process. Furthermore, such systems contain functions that have the same functionality on different datatypes, such as equality functions, print functions, parse functions, etc. Software should be written such that the impact of changes to the software is as limited as possible. Polytypic programs are programs that adapt automatically to changing structure, and thus reduce the impact of changes. This effect is achieved by writing programs such that they work for large classes of datatypes.

Consider for example the function `length :: List a -> Int`, which counts the number of values of type `a` in a list. There is a very similar function `length :: Tree a -> Int`, which counts the number of occurrences of `a`'s in a tree. We now want to generalise these two functions into a single function which is not only polymorphic in `a`, but also in the type constructor; something like `length :: d a -> Int`, where `d` ranges over type constructors. We call such functions *polytypic functions* [15]. Once we have a polytypic `length` function, function `length` can be applied to values of any datatype. If a datatype is changed, `length` still behaves as expected. For example, the datatype `List a` has two constructors with which lists can be built: the empty list constructor, and the `Cons` constructor, which prepends an element to a list. If we add a constructor with which we can append an element to a list, function `length` still behaves as expected, and counts the number of elements in a list.

The equality function in ML and Ada and the functions in the classes that can be derived in Haskell are examples of widely used polytypic functions. These functions are automatically generated by the compiler, but the definitions of these functions cannot be given in the languages themselves. In this paper we investigate a language extension with which such functions can be defined in the language. Polytypic functions are useful in many situations; more examples are given in Jeuring and Jansson [16].

A polytypic function can be applied to values of a large class of datatypes, but some restrictions apply. We require that a polytypic function is applied to values of *regular* datatypes. A datatype `D a` is regular if it contains no function spaces, and if the arguments of the datatype constructor on the left- and right-hand side in its definition are the same. The collection of regular datatypes contains all conventional recursive datatypes, such as `Nat`, `List a`, and different kinds of trees. We

Submitted to the 24th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France, January 15–17, 1997.

introduce the class `Regular` of all regular datatypes, and we write: `length :: Regular d => d a -> Int`

Polytypic functions can be defined on a larger class of datatypes, including datatypes with function spaces [9, 28], but we will not discuss these extensions.

1.1 Polymorphism and polytypism

Polytypism differs from both parametric polymorphism and ad-hoc polymorphism (overloading). This subsection explains how.

A traditional polymorphic function such as

```
head  ::  [a] -> a
```

can be seen as a family of functions - one for each instance of `a` as a monomorphic type. There need only be one definition of `head`; the typing rules ensure that values of type `a` are never used. A polymorphic function can be implemented as a single function that works on boxed values.

An ad-hoc polymorphic function such as

```
(+)  ::  Num a => a -> a -> a
```

is also a family of functions, one for each instance in the `Num` class. These instances may be completely unrelated and each instance is defined separately. Helped by type inference a compiler can almost always find the correct instance.

The polymorphism of a polytypic function such as

```
length  ::  Regular d => d a -> Int
```

is somewhere in between parametric and ad-hoc polymorphism. A single definition of `length` suffices, but `length` has different instances in different contexts. Here the compiler generates instances from the definition of the polytypic function and the type in the context where it is used. A polytypic function may be parametric polymorphic, but it need not be: function `sum :: Regular d => d Int -> Int`, which returns the sum of the integers in a value of an arbitrary datatype, is polytypic, but not parametric polymorphic.

1.2 Writing polytypic programs

There exist various ways to implement polytypic programs in a typed language. Three possibilities are:

- using a universal datatype;
- using higher-order polymorphism and constructor classes;
- using a special syntactic construct.

Polytypic functions can be written by defining a universal datatype, on which we define the functions we want to have available for large classes of datatypes. These polytypic functions can be used on a specific datatype by providing translation functions to and from the universal datatype. However, using universal datatypes has several disadvantages: the user has to write all the translation functions, type information is lost in the translation phase to the universal datatype, and type errors can occur when programs are run. Furthermore, different people will use different universal datatypes, which will make program reuse more difficult.

If we use higher-order polymorphism and constructor classes for defining polytypic functions [12, 19], type information is preserved, and we can use current functional languages such as Gofer and Haskell for implementing polytypic functions. However, writing such programs is rather cumbersome: programs become cluttered with instance declarations, and type declarations become cluttered with contexts. Furthermore, it is hard to deal with mutual recursive datatypes.

Since the first two solutions to writing polytypic functions are dissatisfying, we have extended (a subset of) Haskell with a syntactic construct for defining polytypic functions. Thus polytypic functions can be implemented and type checked. We will use the name PolyP both for the extension and the resulting language. Consult the page

```
http://www.cs.chalmers.se/~johanj/polytypism/
```

to obtain a preliminary version of a compiler that compiles PolyP into Haskell (which subsequently can be compiled with a Haskell compiler), and for the latest developments on PolyP.

1.3 PolyP

PolyP is an extension of a functional language that allows the programmer to define and use polytypic functions. The underlying language in this article is a subset of Haskell and hence lazy, but this is not essential for the polytypic extension. The extension introduces a new kind of (top level) definition, the `polytypic` construct, used to define functions by induction over the structure of datatypes. Since datatype definitions can express sum- , product-, parametric- and recursive types, the `polytypic` construct must handle these cases.

PolyP type checks polytypic value definitions and when using polytypic values types are automatically inferred[1]. The type inference algorithm is based upon Jones' theories of qualified types [18] and higher-order polymorphism [20]. The semantics of PolyP is defined

[1] Just as in Haskell, sometimes explicit type annotations are needed to resolve overloading.

by adding type arguments to polytypic functions in a dictionary passing style. We give a type based translation from PolyP to Haskell that uses partial evaluation to completely remove the dictionary values at compile time. Thus we avoid run time overhead for creating instances of polytypic functions.

The compiler for PolyP is still under development, and has a number of limitations. Polytypic functions can only be applied to values of non mutual recursive, regular datatypes with one type argument. Multiple type arguments can be encoded in a single sum-type, but we are working on a more elegant treatment of multiple type arguments. One of PolyP's predecessors (a preprocessor that generated instances of polytypic functions [11]) could handle mutual recursive datatypes, and we hope to port this part of the predecessor to PolyP in the near future. In the future PolyP will be able to handle mutual recursive datatypes with an arbitrary number of type arguments and in which function spaces may occur.

1.4 Background and related work

Polytypic functions are standard in the Squiggol community, see [24, 26, 27]. Generating instances for specific polytypic functions, such as `(==)`, `map`, `cata`, `hylo`, etc. for a given type, is rather simple and has been demonstrated by several authors [3, 8, 11, 13, 31].

Given a number of predefined polytypic functions many others can be defined, and amongst others Jay et al's type system [2, 13], and Jones' type system based on qualified types and higher-order polymorphism [18, 20] can be used to type check expressions in a language with predefined polytypic functions. Our approach differs from these approaches in that we only give two predefined polytypic functions, and we supply a construct to define new polytypic functions by induction over the structure of datatype definitions. This difference is essential for polytypic programming, and can be compared with the difference between the first versions of ML that gave a number of predefined datatypes and the later versions of ML that provided a few built in types and a construct for defining user-defined datatypes.

Using a two level language, Sheard and Nelson [30] show how to write well-typed polytypic programs. A polytypic program is obtained by embedding second level type declarations as values in first level computations. The two level language is more powerful than our language, but it is also a much larger extension of common functional programming languages.

Adaptive object-oriented programming [23, 29] is a programming style similar to polytypic programming. In adaptive OOP methods (corresponding to our polytypic functions) are attached to groups of classes (types) that usually satisfy certain constraints (such as being regular). In adaptive OOP one abstracts from constructor names instead of datatype structure. This results in a programming style in which typing plays a much less prominent role than in polytypic programming. However, the resulting programs have very similar behaviour.

1.5 About this paper

We will use Haskell syntax for programs and types in our examples. We will use a backward function arrow in types (and kinds), `b <- a`, as syntactic sugar for `a -> b`.

Section 2 introduces polytypic programming. Section 3 discusses the type inference and checking algorithms used in PolyP. Section 4 gives the semantics of PolyP, and Section 5 shows how to generate code for PolyP programs. Section 6 concludes the paper.

2 Polytypic programming

In this section we will show how to write polytypic programs using PolyP. For an extensive introduction to polytypic programming see Jeuring and Jansson [16].

2.1 Functors for datatypes

To define a polytypic function, we have to be able to define functions by induction over the structure of a datatype. The structure of a datatype is described by means of the *functor* defining the datatype.

Consider the datatype `List a` defined by

```
data List a = Nil | Cons a (List a)
```

Values of this datatype are built by prepending values of type `a` to a list. This datatype can be viewed as the fixed point with respect to the second argument of the datatype `FList a x` defined by

```
data FList a x = FNil | FCons a x
```

The datatype `FList a x` describes the structure of the datatype `List a`. Since we are only interested in the structure of `List a`, the names of the constructors of `FList` are not important. We define *FList* using a conventional notation by removing `FList`'s constructors (writing () for the empty space we obtain by removing `FNil`), replacing `|` with +, and replacing juxtaposition with ×.

$$FList\ a\ x \quad = \quad () + a \times x$$

We now abstract from the arguments a and x in *FList*. Constructor *Par* returns the parameter a (the first argument), and *Rec* returns the recursive parameter x (the second argument). Operators $+$ and $\times$ and the empty product $()$ are lifted.

$$FList \quad = \quad () + Par \times Rec$$

$FList$ is the functor[2] of `List a`.

The datatype `Tree a` is defined by

```
data Tree a = Leaf a | Bin (Tree a) (Tree a)
```

Applying the same procedure as for the datatype `List a`, we obtain the following definition.

$$FTree \quad = \quad Par + Rec \times Rec$$

$FTree$ is the functor of `Tree a`.

We have given functors that describe the structure of the datatypes `List a` and `Tree a`. We have that for each regular datatype there exists a (bi)functor F that describes the structure of the datatype[3].

For our purposes, a functor is a value generated by the following grammar.

$$F ::= f \mid F + F \mid F \times F \mid () \mid Par \mid Rec \mid D@F \mid Con\ \tau$$

where f is a functor variable, D generates datatype constructors and τ in $Con\ \tau$ is a type. The alternative $Con\ \tau$ in this grammar is used in the description of the structure of a datatype that contains constant types such as `Bool`, `Char`, etc. The alternative $D@F$ is used to describe the structure of types that are defined in terms of other user-defined types, such as the datatype of *rose-trees*:

```
data Rose a = Fork a (List (Rose a))
```

The functor we obtain for this datatype is

$$FRose \quad = \quad Par \times (List\ @Rec)$$

2.2 The `polytypic` construct

We introduce a new construct `polytypic` for defining polytypic functions by induction on the structure of a functor:

```
polytypic p :: t = case f of {fi -> ei}
```

where `p` is the name of the value being defined, `t` is its type, `f` is a functor variable, `fi` are functor patterns and `ei` are PolyP expressions. The explicit type in the `polytypic` construct is needed since we cannot in general infer the type from the cases.

The informal meaning is that we define a function that takes a functor (a value describing the structure of a datatype) as its first argument. This function selects the expression in the first branch of the case matching the functor. Thus the polytypic construct is a template for constructing instances of polytypic functions given the functor of a datatype. The functor argument of the polytypic function need not (and cannot) be supplied explicitly but is inserted by the compiler during type inference.

As a running example throughout the paper we take the function `flatten` defined in figure 1. When `flatten`

```
flatten  :: Regular d => d a -> [a]
flatten  =  cata fl

polytypic fl :: f a [a] -> [a] =
  case f of
    g + h -> either fl fl
    g * h -> \(x,y) -> fl x ++ fl y
    ()    -> \x -> []
    Par   -> \x -> [x]
    Rec   -> \x -> x
    d @ g -> concat . flatten . pmap fl
    Con t -> \x -> []

data Either a b  =  Left a | Right b

either :: (a->c) -> (b->c) -> Either a b -> c
either f g (Left x)   =  f x
either f g (Right x)  =  g x
```

Figure 1: The definition of `flatten`

is used on an element of type `Tree a`, the compiler performs roughly the following rewrite steps to construct the actual instance of `flatten` for `Tree`:

$$\mathtt{flatten}_{Tree} \quad \to \quad \mathtt{cata}_{Tree}\ \mathtt{fl}_{FTree}$$

It follows that we need an instance of `cata` on the datatype `Tree a`, and an instance of function `fl` on the functor of `Tree a`. For the latter instance, we use the definition of $FTree$ and the definition of `fl` to transform $\mathtt{fl}_{FTree}$ as follows.

$$\mathtt{fl}_{FTree} \to \mathtt{fl}_{Par+Rec\times Rec} \to \mathtt{either}\ \mathtt{fl}_{Par}\ \mathtt{fl}_{Rec\times Rec}$$

We transform the functions $\mathtt{fl}_{Par}$ and $\mathtt{fl}_{Rec\times Rec}$ separately. For $\mathtt{fl}_{Par}$ we have

$$\mathtt{fl}_{Par} \quad \to \quad \mathtt{\backslash x\ \text{->}\ [x]}$$

and for $\mathtt{fl}_{Rec\times Rec}$ we have

$$\begin{array}{ll} & \mathtt{fl}_{Rec\times Rec} \\ \to & \mathtt{\backslash(x,y)\ \text{->}\ fl}_{Rec}\ \mathtt{x\ \text{++}\ fl}_{Rec}\ \mathtt{y} \\ \to & \mathtt{\backslash(x,y)\ \text{->}\ (\backslash x\ \text{->}\ x)\ x\ \text{++}\ (\backslash x\ \text{->}\ x)\ y} \end{array}$$

[2] In fact, $FList$ is a bifunctor: a functor that takes two arguments; we will use both terms functor and bifunctor for bifunctors.

[3] A datatype can be modelled as the initial algebra in the category of F a-algebras [24], where F is the the functor of the datatype.

The last function can be rewritten into `uncurry (++)`, and thus we obtain the following function for flattening a tree:

$\texttt{cata}_{Tree}$ `(either (\x -> [x]) (uncurry (++)))`

By expanding $\texttt{cata}_{Tree}$ in a similar way we obtain a Haskell function for the instance of `flatten` on `Tree`.

The catamorphism, or generalised fold, on a datatype takes as many functions as the datatype has constructors (combined into a single argument by means of function `either`), and recursively replaces constructor functions with corresponding argument functions. It is a generalisation to arbitrary regular datatypes of function `foldr` defined on lists. We will give the definition of `cata` in the next subsection.

2.3 Basic polytypic functions

In the definition of function `flatten` we used functions like `cata` and `pmap`. This subsection defines these and other basic polytypic functions.

Since polytypic functions cannot refer to constructor names of specific datatypes, we introduce the predefined functions `out` and `inn`. Function `out` is used in polytypic functions instead of pattern matching on the constructors of a datatype. For example `out` on `Tree` is defined as follows:

$\texttt{out}_{Tree}$ `(Leaf x)  =  Left x`
$\texttt{out}_{Tree}$ `(Bin l r)  =  Right (l,r)`

Function `inn` is the inverse of function `out`. It collects the constructors of a datatype into a single constructor function.

```
out  ::  Regular d => d a -> fd a (d a)
inn  ::  Regular d => d a <- fd a (d a)
```

where `fd` abbreviates `FunctorOf d`. `FunctorOf` is a special type constructor that takes a datatype constructor, and returns its functor[4]. It is our main means for expressing the relation between datatypes and functors.

In category theory, a functor is a mapping between categories that preserves the algebraic structure of the category. Since a category consists of objects (types) and arrows (functions), a functor consists of two parts: a definition on types, and a definition on functions. The functors we have seen until now are functions that take two types and return a type. The part of the functor that takes two functions and returns a function is called `fmap`, see figure 2.

Using `fmap` we can define the polytypic version of function map, `pmap`, as follows:

[4] With datatypes as fix-points of functors, `FunctorOf` is the 'unfix'.

```
polytypic fmap ::
  (a -> c) -> (b -> d) -> f a b -> f c d
  = \p r -> case f of
              g + h -> fmap p r -+- fmap p r
              g * h -> fmap p r -*- fmap p r
              ()    -> id
              Par   -> p
              Rec   -> r
              d @ g -> pmap (fmap p r)
              Con t -> id

f -+- g = either (Left . f) (Right . g)
(f -*- g) (x,y)  =  (f x , g y)
```

Figure 2: Definition of `fmap`.

```
pmap :: Regular d => (a -> b) -> d a -> d b
pmap f  =  inn . fmap f (pmap f) . out
```

where `out` takes the argument apart, `fmap` applies `f` to parameters and `(pmap f)` recursively to substructures and `inn` puts the parts back together again.

Function `cata` is also defined in terms of function `fmap`:

```
cata :: Regular d =>
  (FunctorOf d a b -> b) -> (d a -> b)
cata f = f . fmap id (cata f) . out
```

This one-liner, together with the definition of `fmap` is all that is needed to obtain a catamorphism for every regular datatype.

2.4 Catamorphisms on specific datatypes

Since catamorphisms are not only useful when defining polytypic functions, but also when defining functions on specific datatypes, we provide a shorthand notation for creating the function argument to `cata`: {`ci -> ei`}. As an example, consider the following datatype of simple expressions.

```
data Expr a  =  Const a
             |  Add (Expr a) (Expr a)
             |  Mul (Expr a) (Expr a)
```

Function `eval` evaluates an expression.

```
eval  :: Num a => Expr a -> a
eval  =  cata feval
  where feval = { Const -> id
                  Add   -> (+)
                  Mul   -> (*) }
```

Evaluating `eval expr` for some `expr :: Expr a` will result in replacing each constructor in `expr` with its corresponding function.

2.5 More polytypic functions

We can define a polytypic equality function using a polytypic `zip` function:

```
(==) :: (Regular d,Eq a) => d a -> d a -> Bool
a == b = maybe False
               (all (uncurry (==)) . flatten)
               (pzip (a,b))

pzip :: Regular d =>
        (d a,d b) -> Maybe (d (a,b))
fzip :: Bifunctor f =>
        (f a b,f c d) -> Maybe (f (a,c) (b,d))
```

where `maybe`, `all` and `uncurry` are predefined Haskell functions. Function `pzip` is a generalisation of the Haskell function `zip :: [a] -> [b] -> [(a,b)]`. Function `zip` takes a pair of lists to a list of pairs. If the lists are of unequal length (that is their structures are different) the longer list is truncated (replaced by the empty structure). In `pzip` a pair of structures is mapped to `Just` a structure of pairs if the structures are equal, and `Nothing` otherwise, since it is in general impossible to know what 'truncate' or an 'empty structure' means for a type `d a`. Function `pzip` is defined using the nonrecursive variant `fzip`, which is defined by means of the `polytypic` construct.

The evaluation of `a == b` gives `False` if `pzip (a,b)` gives `Nothing` and checks that all pairs in the zipped structure are equal otherwise.

In the next subsection we will use function `separate` which separates a value into its structure and its contents.

```
separate :: Regular d => d a -> (d (),[a])
separate x = (pmap (const ()) x, flatten x)
```

Function `separate` is the central function in Jay's [14] representation of values of shapely types: a value of a shapely type is represented by its structure, obtained by replacing all contents of the value with `()`, and its contents, obtained by flattening the value.

2.6 Polytypic data compression

A considerable amount of internet traffic consists of files that possess structure — examples are databases, html files, and JavaScript programs — and it will pay to compress these structured files. Structure-specific compression methods give much better compression results than conventional compression methods such as the Unix compress utility [1, 32]. For example, Unix compress typically requires four bits per byte of Pascal program code, whereas Cameron [4] reports compression results of one bit per byte Pascal program code.

The basic idea of the structure-specific compression methods is simple: parse the input file into a structured value, separate structure from contents, compress the structure into a bit-string by representing constructors by numbers, and compress the resulting string and the contents with a conventional compression method. For example, suppose we have the following datatype for trees:

```
data Tree a = Leaf a | Bin (Tree a) (Tree a)
```

and a file containing the following tree `t`

```
Bin (Bin (Leaf "bar") (Leaf "car"))
    (Bin (Leaf "far") (Leaf "war"))
```

Separating structure from contents gives:

```
Bin (Bin (Leaf ()) (Leaf ()))
    (Bin (Leaf ()) (Leaf ()))
```

and a list containing the four words `bar`, `car`, `far` and `war`. Assigning `0` to `Leaf` and `1` to `Bin` , the above structure can be represented by `1100100`. This bit-string equals 100 when read as a binary number, and hence this list can be represented by the 100'th ASCII character 'd'. So the tree `t` can be represented by the list of words `[d,bar,car,far,war]`. The tree `t` is stored in 68 bytes, and its compressed counterpart requires 19 bytes. This list can be further compressed using a conventional compression method.

Most authors of program code compression programs [4, 5] observe that this method works for arbitrary structured objects, but most results are based on compressing Pascal programs. To compress JavaScript programs we will have to write a new compression program. It is desirable to have a polytypic data compression program.

The description of the basic idea behind polytypic data compression is translated into a polytypic program `pcompress` as follows. Function `pcompress` takes as argument a description (`concrete_syntax`) of how to print values of a datatype (the abstract syntax).

```
pcompress :: (Regular d,Text a) =>
             Syntax d -> String -> String
pcompress concrete_syntax =
    ccompress
  . structure_compress -*- show
  . separate
  . parse concrete_syntax
```

We will describe each of the new functions above in turn. We will omit the precise definitions of these functions.

Function `parse` is a polytypic function of type

```
parse::Regular d => Syntax d -> String -> d a
```

It takes a description of the concrete syntax of a datatype, and returns a parser for that concrete syntax. It is only defined if the grammar for the concrete syntax satisfies certain properties.

Function `structure_compress` takes a structure, replaces its constructors by numbers, and turns the resulting structure into a string.

```
structure_compress :: d () -> String
```

Function `show :: Text a => a -> String` prints the content list generated by `separate` and, finally, function `ccompress` uses a conventional compression program to compress the pair of strings.

```
ccompress :: (String,String) -> String
```

3 Type inference

Polytypic value definitions can be type checked, and for all other expressions the type can be inferred. This section discusses the type checking and type inference algorithms.

The first subsection introduces the core language without the `polytypic` construct, but with qualified and higher-order polymorphic types. The second subsection extends the core with PolyP in two steps. The third subsection discusses unification in the extended language, and the fourth subsection shows how to type check a polytypic value definition.

3.1 The core language

Our core language is an extension of core-ML with qualified types and higher order polymorphism [20], see figure 3. Each constructor in this language has a superscript denoting its kind. For example, a basic type has kind `*`, and a datatype constructor such as `List` has kind `* -> *`. We call the resulting language QML. The set of constructor constants contains:

```
->, (,), Either  ::  * -> * -> *
```

A program consists of a list of datatype declarations and a binding for `main`.

The typing rules and the type inference algorithm are based on the extensions of the standard rules and algorithm [6] that handle qualified and higher-order polymorphic types, see Jones [18, 20]. Compared to the traditional Hindley-Milner system the type judgements are extended with a set of predicates P. The rules involving essential changes in the predicate set are shown in figure 4. The other rules and the algorithm are omitted. The entailment relation $\Vdash$ relates sets of predicates and is used to reason about qualified types, see [18].

$$
\begin{array}{lcll}
E & ::= & x & \text{variable} \\
 & | & EE & \text{application} \\
 & | & \lambda x.E & \text{abstraction} \\
 & | & \texttt{let}\ Q\ \texttt{in}\ E & \text{let-expression} \\
 & & & \\
Q & ::= & x = E & \text{variable binding} \\
 & & & \\
C^\kappa & ::= & \chi^\kappa & \text{constants} \\
 & | & \alpha^\kappa & \text{variables} \\
 & | & C^{\kappa' \to \kappa} C^{\kappa'} & \text{applications} \\
 & & & \\
\tau & ::= & C^* & \text{types} \\
\rho & ::= & P \Rightarrow \tau & \text{qualified types} \\
\sigma & ::= & \forall t_i^\kappa . \rho & \text{type schemes}
\end{array}
$$

Figure 3: The core language QML

$$
\begin{array}{ll}
(\Rightarrow E) & \dfrac{P \mid \Gamma \vdash e : \pi \Rightarrow \rho \qquad P \Vdash \pi}{P \mid \Gamma \vdash e : \rho} \\[2ex]
(\Rightarrow I) & \dfrac{P, \pi \mid \Gamma \vdash e : \rho}{P \mid \Gamma \vdash e : \pi \Rightarrow \rho}
\end{array}
$$

Figure 4: Some of the typing rules for QML

3.2 The polytypic language extension

The polytypic extension of QML consists of two parts - an extension of the type system and an extension of the expression language. We call the extended QML language polyQML.

3.2.1 Extending the type system

The type system is extended by generalising the unification algorithm and by adding new types, kinds and classes to the initial type environment. The initial typing environment of the language polyQML consists of four components: the typings of the functions `inn` and `out`, the type classes `Regular` and `Bifunctor`, two type constructors `FunctorOf` and `Mu`, and the collection of functor constructors (`+`, `*`, `@`, `()`, `Par`, `Rec` and `Con t`).

- Functions `inn` and `out` were introduced in section 2.3.

  ```
  out  ::  Regular d => d a -> fd a (d a)
  inn  ::  Regular d => d a <- fd a (d a)
  ```

  where `fd` abbreviates `FunctorOf d`. Note that these functions have qualified higher-order polymorphic types.

- The class `Regular` contains all regular datatypes and the class `Bifunctor` contains the functors of all regular datatypes. To reflect this the entailment relation is extended as follows for polyQML:

  $\Vdash$ `Regular D`, for all regular datatypes `D a`
  `Regular d` $\Vdash$ `Bifunctor (FunctorOf d)`

- `FunctorOf` is a type constructor that takes a datatype constructor and represents its functor. Type constructor `Mu` is the inverse of `FunctorOf`: it takes a functor, and represents the datatype that has the functor as structure. As there may be different datatypes with the same structure, we add a second argument of `Mu` to disambiguate types. The type constructor `Mu` is useful when we want to relate similar but different types. `FunctorOf` and `Mu` have the following kinds:

  ```
  FunctorOf  ::  1 -> 2
  Mu         ::  1 <- (2,1)
  ```

  where `1` abbreviates the kind of regular type constructors (`*->*`) and `2` abbreviates the kind of bifunctors (`*->*->*`).

- The functor constructors obtained from the nonterminal F are added to the constructor constants, and have the following kinds:

  ```
  * , +                ::  2 -> 2 -> 2
  @                    ::  1 -> 2 -> 2
  (),Par,Rec,Con t     ::  2
  ```

  Each of these constructors has one rule in the entailment relation of one of the following forms:

  `Bifunctor f,Bifunctor g` $\Vdash$ `Bifunctor(f+g)`
  `Regular d, Bifunctor g` $\Vdash$ `Bifunctor (d@g)`
  $\Vdash$ `Bifunctor Par`

The resulting type system is quite powerful; it can be used to type check many polytypic programs in a context assigning types to a number of basic polytypic functions. But although we can use and combine polytypic functions, we cannot define new polytypic function by induction on the structure of datatypes.

At this point we could choose to add some basic polytypic functions that really need an inductive definition to the typing environment. This would give us roughly the same expressive power as the language given by Jay [13] extended with qualified types. As a minimal example we could add `fmap` to the initial environment:

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

letting us define and type check polytypic functions like `pmap` and `cata`. The type checking algorithm would for example derive `pmap (+1) (Leaf 4) :: Regular Tree => Tree Int`. But it would, at best, be hard to write a polytypic version of a function like `zip`. Adding the `polytypic` construct to our language will make writing polytypic programs much simpler.

$$\frac{\Gamma' = (\Gamma, \gamma), \quad \gamma = (x : \sigma) \qquad P_i \mid \Gamma' \vdash e_i : \{f \mapsto f_i\}\sigma}{P_1, \ldots, P_n \mid \Gamma \vdash \texttt{polytypic}\ x : \sigma = \texttt{case}\ f\ \texttt{of}\ \{f_i \to e_i\} : \gamma}$$

Figure 5: The typing rule for `polytypic`

```
type (f + g) a b   =  Either (f a b) (g a b)
type (f * g) a b   =  (f a b , g a b)
type () a b        =  ()
type Par a b       =  a
type Rec a b       =  b
type (d @ g) a b   =  d (g a b)
type Con t a b     =  t
```

Figure 6: Interpreting functors as type synonyms

3.2.2 Adding the `polytypic` construct

To add the `polytypic` construct, the production for variable bindings in the `let`-expression, Q, is extended with

$$\texttt{polytypic}\ x : \rho = \texttt{case}\ f^{*\to*\to*}\ \texttt{of}\ \{F_i \to E_i\}$$

where f is a functor variable, and F is the nonterminal that describes the language of functors defined in Section 2.1. The resulting language is polyQML. To be able to do the case analysis over a functor, it must be built up using the operators `+`, `*`, `@` and the type constants `()`, `Par`, `Rec` and `Con t`. This is equivalent to being in the class `Bifunctor` and thus the context `Bifunctor f` is always included in the type ρ of a function defined by the `polytypic` construct. (But it need not be given explicitly.)

The typing rules for polyQML are the rules from QML together with the rule for typing the `polytypic` construct given in figure 5. For the notation used, see [18]. Note that the `polytypic` construct is not an expression but a binding, and hence the typing rule returns a binding. The rule is not as simple as it looks - the substitution $\{f \mapsto f_i\}$ replaces a functor variable with a functor interpreted as a partially applied type synonym, see figure 6.

3.3 Unification

The (omitted) typing rule for application uses a unification algorithm to unify the argument type of a function with the type of its argument. The presence of the equalities concerning `Mu` and `FunctorOf` complicate unification.

The unification algorithm we use is an extension of the kind-preserving unification algorithm of Jones [20], which in its turn is an extension of Robinson's well-known unification algorithm. We unify under the equalities

$$\begin{aligned}\texttt{Mu (FunctorOf d,d)} &= \texttt{d} && (1)\\ \texttt{FunctorOf (Mu (f,d))} &= \texttt{f} && (2)\\ \texttt{Mu}(\mathcal{F}_{\texttt{D}}\texttt{,D}) &= \texttt{Mu(FunctorOf D,D)} && (3)\end{aligned}$$

where $\mathcal{F}_{\texttt{D}}$ is the functor corresponding to the datatype `D a` built with the functor constructors. The last equality represents a set of equalities: one such equality is generated for each regular datatype declared in the program. For example, if a program declares the datatype `List a`, the equality

```
Mu(()+Par*Rec,List) = Mu(FunctorOf List,List)
```

is generated.

We will write $C \sim^{\sigma} C'$ if C and C' are unified under equalities (1), (2), and (3) by substitution σ. For example, we have

$$\begin{aligned}\texttt{Tree a} &\sim^{\sigma_1} \texttt{Mu (f,d) a}\\ \texttt{Mu (f + g,d)} &\sim^{\sigma_2} \texttt{Mu (FunctorOf List,List)}\end{aligned}$$

$$\text{where} \begin{cases}\sigma_1 = \{\texttt{f} \mapsto \texttt{FunctorOf Tree}, \texttt{d} \mapsto \texttt{Tree}\}\\ \sigma_2 = \{\texttt{f} \mapsto \texttt{()}, \texttt{g} \mapsto \texttt{Par*Rec}, \texttt{d} \mapsto \texttt{List}\}\end{cases}$$

Unification under equalities is known as semantic unification, and is considerably more complicated than syntactic unification. In fact, for many sets of equalities it is impossible to construct a (most general) unifier. However, if we can turn the set of equalities under which we want to unify into a complete (normalising and confluent) set of rewriting rules, we can use one of the two algorithms (using narrowing or lazy term rewriting) from Martelli et al. [21, 25] to obtain a most general unifier for terms that are unifiable.

If we replace the equality symbol by $\rightarrow$ in our equalities, we obtain a complete set of rewriting rules. We use the recursive path orderings technique as developed by Dershowitz [7, 21] to prove that the rules are normalising, and we use the Knuth-Bendix completion procedure [21, 22] to prove that the rules are confluent. Both proofs are simple.

Theorem. If there is a unifier for two given types C, C', then $C \sim^{\sigma} C'$ using Jones [20] for kind-preserving unification and Martelli et al's [25] algorithm for semantic unification, and σ is a most general unifier for C and C'. Conversely, if no unifier exists, then the unification algorithm fails.

$$\frac{\begin{array}{c}\Gamma' = (\Gamma, \gamma), \quad \gamma = (x : \forall_{\{\}}(\rho))\\ P_i \mid S_i(T_{i-1}\Gamma') \vdash^{w} e_i : \tau_i\\ \forall_{T_n\Gamma'}(S_n \cdots S_{i+1}(P_i \Rightarrow \tau_i)) \geq \{f \mapsto f_i\}\rho\\ T_0 = \{\}, \quad T_i = S_i T_{i-1}\end{array}}{\Gamma \vdash^{w} \texttt{polytypic}\ x : \rho = \texttt{case}\ f\ \texttt{of}\ \{f_i \rightarrow e_i\} : \gamma}$$

Figure 7: The alternative for `polytypic` in $\mathcal{W}$

3.4 Type checking the `polytypic` construct

Instances of polytypic functions generated by means of a function defined with the `polytypic` construct should be type correct. For that purpose we type check polytypic functions.

Type checking a polytypic value definition amounts to checking that the inferred types for the case branches are more general than the corresponding instances of the explicitly given type. So for each polytypic value definition `polytypic` $x : \rho =$ `case` f `of` $\{f_i \rightarrow e_i\}$ we have to do the following for each branch of the case:

- Infer the type of $e_i : \tau_i$.
- Calculate the type the alternative should have according to the explicit type: $\rho_i = \{f \mapsto f_i\}\rho$.
- Check that ρ_i is an instance of τ_i.

When calculating the types of the alternatives the functor constructors are treated as type synonyms defined in figure 6. The complete type inference/checking algorithm $\mathcal{W}$ is obtained by extending Jones' type inference algorithm [20] with the alternative for the `polytypic` construct given in figure 7. As an example we will sketch how the definition of `fl` in figure 1 is type checked:

In the `g*h` branch of the polytypic case, we first infer the type of the expression $e_* =$ `\(x,y) -> fl x ++ fl y`. Using fresh instances of the explicit type $\rho =$ `f a [a] -> [a]` for the two occurrences of `fl` we get $\tau_* =$ `(x b [b],y b [b]) -> [b]`. We then calculate the type $\rho_* = \{\texttt{f} \mapsto \texttt{g*h}\}\rho =$ `(g*h) a [a] -> [a]` $=$ `(g a [a],h a [a]) -> [a]`. Since $\rho_* = \{\texttt{x} \mapsto \texttt{f}, \texttt{y} \mapsto \texttt{g}, \texttt{b} \mapsto \texttt{a}\}\tau_*$ we see that ρ_* is an instance of τ_*.

In the `Rec` branch of the polytypic case, we first infer the type of the expression $e_{Rec} =$ `\x -> x`. The type of this expression is $\tau_{Rec} =$ `b -> b`. We then calculate the type $\rho_{Rec} = \{\texttt{f} \mapsto \texttt{Rec}\}\rho =$ `Rec a [a] -> [a]` $=$ `[a] -> [a]`. Since $\rho_{Rec} = \{\texttt{b} \mapsto \texttt{[a]}\}\tau_{Rec}$ we see that ρ_{Rec}

is an instance of τ_{Rec}. The other branches are handled similarly.

If a `polytypic` binding can be type checked using the typing rules, algorithm $\mathcal{W}$ also manages to type check the binding. Conversely, if algorithm $\mathcal{W}$ can type check a `polytypic` binding, then the binding can be type checked with the typing rules too. Together with the results from Jones [18] we obtain the following theorem.

Theorem. The type inference/checking algorithm is sound and complete.

4 Semantics

The meaning of a QML expression is obtained by translating the expression into a version of the polymorphic λ-calculus called QP that includes constructs for evidence application and evidence abstraction. Evidence is needed in the code generation process to construct code for functions with contexts. For example, if the function `(==)` of type $\forall$`a . Eq a => a -> a -> Bool` is used on integers somewhere, we need evidence for the fact `Eq Int`, meaning that `Int` has an equality. One way to give evidence in this case is simply to supply the function `primEqInt`. Again, the results from this section are heavily based on Jones' work on qualified types [18].

The language QP has the same expressions as QML plus two new constructions:

$$
\begin{array}{lcll}
E & ::= & \cdots & \text{same as for QML expressions} \\
 & | & E_e & \text{evidence application} \\
 & | & \lambda v.E & \text{evidence abstraction} \\
 & & & \\
\sigma & ::= & C^* & \text{types} \\
 & | & P \Rightarrow \sigma & \text{qualified types} \\
 & | & \forall t_i^\kappa.\sigma & \text{polymorphic types}
\end{array}
$$

For notational convenience we will also use `case`-statements. The typing rules for QP are omitted.

Except for the translation rule for the `polytypic` construct given in figure 8, the translation rules are simple and omitted. A translation rule of the form $P \mid S(\Gamma) \vdash^w e \leadsto e' : \tau$ can be read as an attribute grammar. The inherited attributes (the input data) consist of a type context Γ and an expression e and the synthesised attributes (the output data) are the evidence context P, the substitution S, the translated QP expression e' and the inferred type τ.

For example, if we translate function `fl :: Bifunctor f => f a [a] -> [a]`, we obtain after simplification the code in figure 9.

$$
\begin{array}{lll}
\texttt{fl} = & \multicolumn{2}{l}{\lambda v.\texttt{case}\ v\ \texttt{of}} \\
 & f + g & \rightarrow\ \texttt{either fl}_f\ \texttt{fl}_g \\
 & f * g & \rightarrow\ \texttt{\textbackslash(x,y) -> fl}_f\ x\ \texttt{++ fl}_g\ y \\
 & () & \rightarrow\ \texttt{\textbackslash x -> []} \\
 & Par & \rightarrow\ \texttt{\textbackslash x -> [x]} \\
 & Rec & \rightarrow\ \texttt{\textbackslash x -> x} \\
 & d\ @\ g & \rightarrow\ \texttt{concat . flatten}_d\ \texttt{. pmap}_d\ \texttt{fl}_g \\
 & Con\ t & \rightarrow\ \texttt{\textbackslash x -> []}
\end{array}
$$

Figure 9: The translation of function `fl` into QP

In this translation we use a conversion function C, which transforms evidence abstractions applied to evidence parameters into an application of the right type. Function C is obtained from the expression $\sigma \geq^C \sigma'$, which expresses that σ is more general than σ' and that a witness for this statement is the conversion function $C : \sigma \rightarrow \sigma'$. The inputs to function $\geq$ are the two type schemes σ and σ', and the output (if it succeeds) is the conversion function C. It succeeds if the unification algorithm succeeds on the types and the substitution is from the left type to the right type only, and if the evidence for the contexts in σ can be constructed from the evidence for the contexts in σ'. The function C is constructed from the entailment relation extended with evidence values.

As evidence for the fact that a functor `f` is a bifunctor we take a symbolic representation f of the functor (an element of the datatype described by nonterminal F from Section 2.1). So f : `Bifunctor f` for all `f` for which $\Vdash$ `Bifunctor f` holds. The evidence for regularity of a datatype `D a` is a dictionary with three components: the definitions of `inn` and `out` on the datatype and evidence that the corresponding functor is indeed a bifunctor.

Theorem. The translation from polyQML to QP preserves well-typedness and succeeds for programs with unambiguous type schemes.

5 Code generation

To generate code for a polyQML program, we generate a QML expression from a polyQML expression in two steps:

- A polyQML expression is translated to a QP expression with explicit evidence parameters (dictionaries).
- The QP expression is partially evaluated with respect to the evidence parameters giving a program in QML.

$$\frac{\begin{array}{c}\Gamma' = (\Gamma, \gamma), \quad \gamma = (x : \forall_{\{\}}(\rho)) \\ v_i : P_i \mid S_i(T_{i-1}\Gamma') \vdash^w e_i \leadsto e_i' : \tau_i \\ \forall_{T_n\Gamma'}(S_n \cdots S_{i+1}(P_i \Rightarrow \tau_i)) \geq^{C_i} \forall_{\{\}}(\{f \mapsto f_i\}\rho) \\ T_0 = \{\}, \quad T_i = S_i T_{i-1}\end{array}}{\Gamma \vdash^w (\texttt{polytypic}\ x : \rho = \texttt{case}\ f\ \texttt{of}\ \{f_i \to e_i\}) \leadsto (x = \lambda v.\texttt{case}\ v\ \texttt{of}\ \{f_i \to C_i(\lambda v_i.e_i')v\}) : \gamma}$$

Figure 8: The translation of the `polytypic` construct

When the program has been translated to QP all occurrences of the `polytypic` construct and all references to the classes `Regular` and `Bifunctor` have been removed and the program contains evidence parameters instead. We remove all evidence parameters introduced by polytypism by partial evaluation [17]. The partial evaluation is started at the `main` expression (which must have an unambiguous type) and is propagated through the program by generating requests from the `main` expression and its subexpressions.

The evidence for regularity of a datatype `D a` (the entailment $\Vdash$ `Regular D`) is a dictionary containing the functions `inn`, `out` and the bifunctor $\mathcal{F}_{\mathtt{D}}$. PolyP constructs these dictionaries with a number of straightforward inductive functions over the abstract syntax of regular datatypes. Functions `inn` and `out` are now obtained by selecting the correct component of the dictionary.

In practice, a PolyP program (a program written in a subset of Haskell extended with the polytypic construct) is compiled to Haskell (Hugs). In the appendix we have given an example PolyP program and the code that is generated for this program.

If the size of the original program is n, and the total number of subexpressions of the bifunctors of the regular datatypes occurring in the program is m, then the size of the generated code is at most $n \times m$. Each request for an instance of a function defined by means of the polytypic construct on a datatype `D a` results in as many functions as there are subexpressions in the bifunctor `f` for datatype `D a` (including the bifunctors of the datatypes used in `f`). The efficiency of the generated code is only a constant factor worse than hand-written instances of polytypic functions.

6 Conclusions and future work

We have shown how to extend a functional language with the `polytypic` construct. The `polytypic` construct considerably simplifies writing programs that have the same functionality on a large class of datatypes (polytypic programs). The extension is a small but powerful extension of a language with qualified types and higher-order polymorphism. We have developed a compiler that compiles Haskell with the `polytypic` construct to plain Haskell.

A lot of work remains to be done. The compiler has to be extended to handle mutual recursive datatypes with an arbitrary number of type arguments and in which function spaces may occur. For example, for the purpose of multiple type arguments we will introduce a class `Functor n`, with `Regular` = `Functor 1`, and `Bifunctor` = `Functor 2`. The constructors `Mu` and `FunctorOf` have to be extended in a similar fashion.

The partial evaluation approach to code generation implies that we cannot compile a module containing a definition of a polytypic function separately from a module in which it is used. A solution might be to translate polytypic programs into a language with intensional polymorphism [10] instead of translating polytypic programs into QP.

Acknowledgements
The discussions on type systems for polytypic programming with Graham Hutton, Mark Jones, Oege de Moor, Rinus Plasmeijer, Fritz Ruehr, Tim Sheard and the comments from an anonymous referee are gratefully acknowledged. Oege de Moor, Graham Hutton and Arjan van IJzendoorn helped implementing predecessors of PolyP.

References

[1] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression*. Prentice Hall, 1990.

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

[3] C. Böhm and A. Berarducci. Automatic synthesis of type λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[4] Robert D. Cameron. Source encoding using syntactic information source models. *IEEE Transactions on Information Theory*, 34(4):843–850, 1988.

[5] J.F. Contla. Compact coding of syntactically correct source programs. *Software–Practice and Experience*, 15(7):625–636, 1985.

[6] L. Damas and R. Milner. Principal type-schemes for functional programs. In *9th Symposium on Priciples of Programming Languages, POPL '82*, pages 207–212, 1982.

[7] N. Dershowitz. A note on simplification orderings. *Information Processing Letters*, 9(5):212–215, 1979.

[8] J.H. Fasel, P. Hudak, S. Peyton Jones, and P. Wadler. Sigplan Notices Special Issue on the Functional Programming Language Haskell. *ACM SIGPLAN notices*, 27(5), 1992.

[9] P. Freyd. Recursive types reduced to inductive types. In *Proceedings Logic in Computer Science, LICS '90*, pages 498–507, 1990.

[10] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *22nd Symposium on Priciples of Programming Languages, POPL '95*, pages 130–141, 1995.

[11] P. Jansson. Polytypism and polytypic unification. Master's thesis, Chalmers University of Technology and University of Göteborg, 1995.

[12] P. Jansson and J. Jeuring. Polytypic unification — implementing polytypic functions with constructor classes. In preparation, see `http://www.cs.chalmers.se/~johanj`, 1996.

[13] C. Barry Jay. Polynomial polymorphism. In *Proceedings of the Eighteenth Australasian Computer Science Conference*, pages 237–243, 1995.

[14] C. Barry Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.

[15] J. Jeuring. Polytypic pattern matching. In *Conference Record of FPCA '95, SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture*, pages 238–248, 1995.

[16] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Proceedings of the Second International Summer School on Advanced Functional Programming Techniques*, pages 68–114. Springer-Verlag, 1996. LNCS 1129.

[17] Mark P. Jones. Dictionary-free overloading by partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Orlando, Florida, June 1994.

[18] Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, 1994.

[19] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, LNCS 925, pages 97–136. Springer-Verlag, 1995.

[20] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, pages 1–35, 1995.

[21] J.W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science*, pages 1–116. Oxford University Press, 1992.

[22] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[23] K.J. Lieberherr, I. Silva-Lepe, and C. Xiao. Adaptive object-oriented programming — using graph-based customization. *Communications of the ACM*, pages 94–101, 1994.

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

[25] A. Martelli, C. Moiso, and C.F. Rossi. An algorithm for unification in equational theories. In *Proc. Symposium on Logic Programming*, pages 180–186, 1986.

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

[27] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes, and barbed wire. In J. Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA '91*, pages 124–144, 1991.

[28] E. Meijer and G. Hutton. Bananas in space: Extending fold and unfold to exponential types. In *Conference Record of FPCA '95, SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture*, pages 324–333, 1995.

[29] J. Palsberg, C. Xiao, and K. Lieberherr. Efficient implementation of adaptive software. *TOPLAS*, 1995.

[30] T. Sheard and N. Nelson. Type safe abstractions using program generators. Unpublished manuscript, 1995.

[31] Tim Sheard. Automatic generation and use of abstract structure operators. *ACM Transactions on Programming Languages and Systems*, 13(4):531–557, 1991.

[32] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

A Appendix

A.1 A simple PolyP program

Combining the definitions from figure 1 (`flatten`) with the definition of `fmap` in figure 2 and the code below we get a small polytypic program testing the function `separate`. We assume a prelude containing composition and definitions of the functions `const`, `concat`.

```
main = (separate l,separate r)

l = Cons 1 (Cons 2 Nil)
r = Fork 1 (Cons (Fork 2 Nil) Nil)

data List a = Nil | Cons a (List a)
data Rose a = Fork a (List (Rose a))

separate x = (pmap (const ()) x,flatten x)

pmap f = inn . fmap f  (pmap f) . out

cata h =  h  . fmap id (cata h) . out
```

A.2 The generated code

The code generated by PolyP looks as follows. We have edited the generated code slightly.

```
uncurry0 f p = f
uncurry2 f p = f (fst p) (snd p)

data List a = Nil | Cons a (List a)
data Rose a = Fork a (List (Rose a))

main = (separate_f4List l, separate_f4Rose r)
l = Cons 1 (Cons 2 Nil)
r = Fork 1 (Cons (Fork 2 Nil) Nil)
separate_f4List x =
 (pmap_f4List (const ()) x, flatten_f4List x)
separate_f4Rose x =
 (pmap_f4Rose (const ()) x, flatten_f4Rose x)
pmap_f4List f = inn_f4List
              . fmap_f4List f (pmap_f4List f)
              . out_f4List
pmap_f4Rose f = inn_f4Rose
              . fmap_f4Rose f (pmap_f4Rose f)
              . out_f4Rose
flatten_f4List = cata_f4List fl_f4List
flatten_f4Rose = cata_f4Rose fl_f4Rose
inn_f4List = either (uncurry0 Nil)
                    (uncurry2 Cons)
fmap_f4List =
   \p r -> (fmap_e p r) -+- (fmap_Ppr p r)
out_f4List x = case x of
                 Nil -> Left ()
                 (Cons a b) -> Right (a, b)
cata_f4List h = h
              . fmap_f4List id (cata_f4List h)
              . out_f4List
fl_f4List = either fl_e fl_Ppr
inn_f4Rose = uncurry2 Fork
fmap_f4Rose =
  \p r -> (fmap_p p r) -*- (fmap_A4Listr p r)
out_f4Rose x = case x of
                 (Fork a b) -> (a, b)
cata_f4Rose h = h
              . fmap_f4Rose id (cata_f4Rose h)
              . out_f4Rose
fl_f4Rose = \(x, y) -> fl_p x ++ fl_A4Listr y
f -+- g = either (Left . f) (Right . g)
fmap_e = \p r -> id
fmap_Ppr = \p r -> fmap_p p r -*- fmap_r p r
fl_e = \x -> ([])
fl_Ppr = \(x, y) -> (fl_p x) ++ (fl_r y)
(f -*- g)  (x, y) = (f x, g y)
fmap_p = \p r -> p
fmap_A4Listr = \p r-> pmap_f4List (fmap_r p r)
fl_p = \x -> x : ([])
fl_A4Listr = concat . flatten_f4List
           . (pmap_f4List fl_r)
fmap_r = \p r -> r
fl_r = \x -> x
```