Parsers, Pretty Printers and PolyP

Staffan L. Björk

September 17, 1997

Master of Science Thesis in Computer Science, Göteborg University

email: md2bjork@mdstud.chalmers.se

Abstract

Generating parsers is a common problem in computer science. The automatic generation of parsers reduces time and energy in most larger computer projects. This paper describes one way to generate a parser given a special kind of pretty printer using the polytypic programming language PolyP. This is useful in most programming situations as pretty printers are easier to write than parsers. The parser constructed in this paper is a polytypic function, and this implies that a parser can be generated within the program in which it is used, avoiding a separate generation step.

Polytypic functions can be written in PolyP, an extension of the functional language Haskell developed at the department of Computer Science at Göteborg University and the Technical University of Chalmers.

keywords: Algebraic datatypes, Functional programming, Parsing, Polytypic programming

Sammanfattning

Att skriva en parser är ett vanligt problem inom datavetenskap. Att automatiskt kunna skapa en parser skulle innebära en besparing i både tid och energi i många programmeringsprojekt. Detta examensarbete beskriver ett sätt att generara en parser givet en speciell sorts pretty printer med hjälp av det polytypiska programmingsspråket PolyP. Detta skulle underlätta programmeringen eftersom det är lättare att skriva en pretty printer än en parser. Den parsergenerator som beskrivs här skiljer sig från övriga parser generatorer genom att parsern konstrueras inuti programet, vilket gör att man slipper ett separat kompileringssteg för parsern.

Polytypiska funktioner kan skrivas i en utbyggnad av det funktionella språket Haskell kallat PolyP, som har utvecklats på Göteborgs Universitet och Chalmers Tekniska Högskola.

1 Preface

This thesis has been written at the Department of Computer Science at Göteborg University and Chalmers University of Technology as a Master of Science thesis during the autumn of 1996 and spring of 1997. The programming was done at the Department of Computer Science using PolyP 0.2-0.4, Hugs 1.3 and hbc 1.3 (the Chalmers haskell compiler). Johan Jeuring and Patrik Jansson, who developed the PolyP language, were the supervisors of the thesis, and taught me how to write polytypic programs. Without their help this thesis would not have been written. They also gave considerable help in making the language in the thesis correct and understandable.

The working method used in this thesis can be described as either demand driven or problem solving. After the first decision on the topic of the master's thesis, Johan Jeuring and I examined the problem from a general perspective in order to obtain an overview of the problem area. I was then given a smaller part to examine. Whenever I got stuck with a technical or theoretical detail, or felt that I had solved a part of the problem, a new discussion took place between myself and my supervisors. If we agreed that things were going in the right direction, I was given a new part to look at. This way of working made me feel active in my learning; i.e. I was not being told to read such and such examples and methods and then apply them to my problem, but instead the course of the thesis was determined and changed by my own findings along the way.

I also believe that this way of working has given me a deeper understanding of the problems that I have encountered and how research is being done at the department. I feel that I have put in a personal effort solving the problems and creating something satisfactory for myself.

2 Introduction

Nearly every computer program that is written is interactive in the sense that it takes input from some user and then presents an output to that user. The input that the program receives needs to be given in an orderly and predetermined fashion so that the program can store the information in an internal data structure. The way the input must be presented is defined by a *grammar*.

As an interactive program can receive different input each time it is run, it must be able to recognise a structure in the input and translate it to the form used internally in the program. This is called *parsing*. As parsing is a very common problem in computer science, much work has been done to automate the creation of a parser by using *parser generators*. The traditional way these generators work is that the programmer gives the grammar for the input and the parser generator gives a function that recognises the structure of the input. The programmer then manually has to specify the translation to the internal structure.

This thesis takes a somewhat different approach by using an extension of the functional programming language Haskell called *PolyP*. This allows the use of *polytypic functions* which makes it possible to generate a parser from the function that is used to construct a string representation for a value of an abstract syntax. This function is called a *pretty printer* and is very often needed in programs and is much easier to write than parsers.

The parser generator described in this thesis contains a parser and it will be shown how that parser is constructed. This also shows how one produces parsers manually. The functions in the program uses *monads* and *locally quantified datatypes*[1] which will also be described.

2.1 Background

Considerable work has been done on the construction of parser generators using imperative languages, mainly C. Two of the most popular ones are Yacc and Bison (even though Yacc is presented as yet another compiler compiler). Both these generators create LALR(1) parsers in the form of C program code. Bison is part of the GNU project and more information is available from the GNU Project [6]. Recently, work has been done to create parser generators in Java [10, 14, 2]. The systems produce LL(1) (the first two) and LALR (the last one) parsers.

2.2 User-defined datatypes

Many programming languages allow a programmer to define new datatypes based on other types[2]. The lazy functional programming language Haskell is one such

[1] At the moment, PolyP cannot handle locally quantified datatypes which means that some minor modifications have to be made by hand after the compilation.

[2] This paper deals only with parsers that parse user-defined datatypes based on one other type. This is due to a limitation of PolyP.

language, and will be used throughout the paper. For example, a user may need a list of some other datatype, and want the list to be possibly infinite in length. A list is either empty or contains an element of some value and the rest of the list. This can be achieved by declaring the datatype `List a` in Haskell, where `a` can be an arbitrary datatype[3].

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

Another common datatype is `Tree a`, which is similar to `List` but it branches at every internal node.

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

Datatypes can also use previously declared datatypes, as in the case of the datatype `Rose a`. A rose tree (as it is called) consists of a pair where the first value is of type `a` and the second value is a list of rose trees (of type `a`).

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

In the rest of this paper, these datatypes will be used in examples of both pretty printers and parsers.

2.3 Parsers and Pretty Printers

Two functions that are often needed for a datatype when programming are the transformation of a `String` to a datatype and the transformation of that datatype to a `String`. This is because strings are often used for both input and output. In the case of `List a`, these functions are normally called `list2string` and `string2list`. The first type of function is a pretty printer (or ugly printer) and the second is a parser. The types of these functions on the datatype `List a` are:

- `list2string :: List a -> String`
- `string2list :: [String] -> Return (List a) [String]`

The generalized type for these functions[4] are:

- `type Pretty_printer d a = d a -> String`
- `type Parser b d a = [b] -> Return (d a) [b]`

[3] In the paper datatypes will appear both normal (List) and typewriter (`List`) fonts. The typewriter font indicates a user-defined datatype in Haskell while a type in normal font stands for the "general" version of the datatype used in computer science.

[4] These will be explained in more detail in chapters 3 and 4 respectively. It will also be explained why `b` and not `String` is used.

Here the datatype `List` is replaced with the datatype variable `d` which can be any user-defined datatype with one type argument. The type `Return` can be seen as an abstract type, as its actual implementation can be a number of datatypes. However, it must be able to allow failures, as for example `Maybe (d a)` does. This is because a parser is not a one-to-one function from `String -> d a`. There might be zero, one or more ways to parse a string. In case there is more than one correct answer, only the first answer is returned.

A value of type `Maybe a` can either be `Nothing`, which indicates a failure, or `Just x` which indicates a success and that the result of the function is the value `x`. This makes it a suitable candidate for the implementation of the abstract datatype and it is used in the thesis.

Pretty printers are usually much easier to write than parsers, and both are very often used. One would therefore like to have a function which takes a pretty printer as an argument and returns a parser. This would give the following function:

- `parser :: Pretty_printer d a -> Parser String d a`

It is possible to write such a function but for each datatype one would have to write a separate function as well as a pretty printer. And writing the function that generates a parser is more complicated than writing the parser itself. But what if one could write a function `parser` which could be used for every datatype? Then one would only have to write a pretty printer for a datatype and get a parser for free.

2.4 PolyP

Patrik Jansson and Johan Jeuring have developed a precompiler called PolyP for Haskell which translates programs with polytypic functions into pure Haskell code [13]. Polytypic functions are functions that can analyse the underlying structure of the datatype of its argument. They do this by using methods from category theory, using the information from the analysis to determine their behaviour. PolyP's main functionality is typechecking the polytypic functions in the program being compiled and generating the instances of the polytypic functions that are used.

It is the goal of this paper to show how polytypic programming and especially PolyP can be used to produce functions which convert pretty printers to parsers. This is done by introducing an intermediate datatype, `Struct a`, and demanding that pretty-printers have the type: `d a -> Struct a`[5]. The datatype `Struct` is used instead of usual `String` in the pretty printers as it preserves information about the syntax of the datatype that is pretty printed. `Struct` is an two-layered construction. The top layer deals with the order of the information and the bottom layer, `Value a`, is used to store the actual data[6]. Using this datatype, it is possible to create two functions `string2struct` and `struct2d` which used together are a parser given two arguments.

[5]This is from now on the type of a pretty printer unless noted otherwise.

[6]The definition of the datatype `Struct a` and `Value a` can be found in section 3.1.

2.5 Other related work

Marieke Huisman from Universiteit Utrecht has done work on generating the kind of parsers discussed here [9]. She has created an algorithm which given a datatype can produce a parser. Only direct string representations of datatype values can be parsed, more sophisticated string representations cannot be handled. Although she has proven the correctness of the algorithm, it is not yet implemented as a program.

Haskell itself can generate parsers and pretty printers by using the *deriving* form for the classes `Read` and `Show`. The class `Show` is used by the pretty printers in section 7. Information about Haskell can be found at `http://www.haskell.org/`.

The Derive preprocessor [3] allows derivation of classes not supported by the standard Haskell compilers. The user may add new rules to the system, allowing the creation of more complex parsers than in standard Haskell. Unfortunately, Derive does not type-check the variables used in the classes.

3 Pretty Printers

The basic idea for a pretty printer is to make a representation of a value of a datatype so that a human can understand it. The simplest pretty printer just translates a value into an unformatted string, but more advanced versions make the values easier to read by using indentation or other methods. Some of the techniques for pretty printing values of datatypes are discussed in [8].

3.1 Examples

A simple way of pretty printing a value of the datatype `List a` is:

```
pp_L :: Show a => List a -> String
pp_L Nil          = "Nil"
pp_L (Cons x xl) = "Cons " ++ (show x) ++ (pp_L xl)
```

This just concatenates the names of the constructors and the values and produces a string with no parentheses or line breaks. It also requires that Haskell knows how to pretty print values of the datatype `a`.

Often, a pretty printer should not show every constructor and value used in the value to be printed. To show everything does not add clarity but instead hides the interesting data in unimportant details. For example, when the datatype `Struct a` is used as an intermediate datatype when pretty printing, then the constructors in `Struct a` should not be shown when pretty printing `Struct a`. The pretty printer `struct2string` is an example of a pretty printer that does not show the constructors of the value it is pretty printing:

```
data Value a = Par a
             | Rec (Struct a)
```

```
              | Comp [?d (Struct a)]
                     (?d (Struct a) -> Struct (Struct a))
                     (Struct (Struct a)) deriving Show

data Struct a = Arg Int (Value a)
              | Symb [Char]
              | Beside (Struct a) (Struct a) deriving Show


value2string :: Show a => Value a -> String
value2string (Par a)      = show a
value2string (Rec a)      = struct2string a
value2string (Comp c f a) = "( " ++ struct2string a ++ " )"

struct2string :: Show a => Struct a -> String
struct2string (Arg n a)    = value2string a
struct2string (Symb a)     = a
struct2string (Beside x y) = struct2string x ++ " " ++
                                struct2string y

instance Show a => Show (Struct a) where
 showsPrec p x = showString (struct2string x)
```

The instance declaration above tells Haskell how to present values of the type `Struct` to the user (by showing the string generated by `struct2string`). This is needed for the `Comp` case in the function `value2string` as the parameter `a` is of the type `struct2string`.

4 Parsers

This description of parsers is based on a paper by Niklas Röjemo called *Efficient parsing combinators* [22]. The description only deals with aspects of parsing relating to the main subject of the paper, i.e. it will only deal with functional monadic parsers and the code related to the positioning of the input has been removed to add clarity. The code for the parser operators is given in appendix B and these are used to solve the main problem of the thesis. Another source for descriptions of monadic parsers is *Monadic Parser Combinators* [5].

Parsing is the act of recognising a structure and transforming data (usually a string) to another form a data. In the process of parsing, extra information can be included in the data produced or redundant information can be removed. The redundancy in the original form can, to take only two examples, be because of information to enhance readability or to specify precedences which will no longer be needed in the new form.

As mentioned in section 2.3, the type of a parser is:

```
type Parser b d a = [b] -> Return (d a) [b]
```

The type variables used by the parser are the type of the values of the list in the input and the type of of the value parsed when the parsing succeeds. A standard method of parsing takes a list of tokens (usually represented as a list of strings) and returns a value that is either a success or a failure. Hence the type given above.

4.1 The lexical analyser

To ease the work of the parser, a lexical analyser first breaks down the input into a number of tokens. This makes it easier at a later stage to distinguish between strings of text, keywords and numbers. As an example, the lexical analyser gives the following list of tokens when give the string `"foo bar3"`:

```
["foo", "bar", "3"]
```

The code for the lexical analyser is given in appendix B.

4.2 Parser combinators

Many parsers can be constructed by using a number of parser combinators. These functions return different parsers depending on which input is received. More complex parser can then be constructed by other parser combinators which construct new parsers by using already existing ones.

```
failure :: Parser a b
succeed :: b -> Parser a b
satisfy :: (a -> Bool) -> Parser a a
into :: Parser a b -> (b -> Parser a c) -> Parser a c
alt :: (Parser a b) -> (Parser a b) -> (Parser a b)
ap :: Parser a (b->c) -> Parser a b -> Parser a c
```

The function `failure` is a parser that immediately fails and `succeed` is a parser that always succeeds in producing an answer without actually consuming any input. The parser `satisfy` takes a test function and only succeeds in parsing if the input value passes the test. The parser `into` takes two arguments of which the first is a parser and the second is a function which produces a parser given a value of the same type as the return type of the parser in the first argument. When `into` is applied, it tries to parse using its first argument and if it succeeds gives the result to the second argument and then uses the parser produced to parse the next token. The function `alt` takes two parsers and first tries the first parser, and if that fails it tries the second parser and returns its value, otherwise it just returns the value produced from the first parser. Finally, `ap` takes two parsers: one which produces a function if it succeeds in parsing a value, another which is an "ordinary" parser. If the "ordinary" parser succeeds in parsing a value, its result is given to the result of the other parser.

4.3 Examples of parsers

Given these operators, it is possible to construct a number of primitive parsers. These parsers are often used as building blocks for other parsers. The `number` parser takes a list of tokens and tries to parse a number from the first token. The `literal` parser checks the first token of a list (which in this case does not have to be a `String`) against a value and succeeds with parsing if the token is equal to the value. Finally, the `match` parser compares a list of tokens against the input, succeeding if the input begins with the same list as the token list.

```
number :: Parser String Int
number = succeed read `ap` satisfy (all isDigit)

literal :: Eq a => a -> Parser a a
literal p = satisfy (p==)

match :: Eq a => [s] -> Parser s ()
match []     = succeed ()
match (t:ts) = literal t `into` \_ -> match ts
```

A more advanced parser is given in section 8.3.

5 Monads

Monads, an idea that first appeared in category theory[7], are often used in functional programming (see [20, 21, 24, 25]). This programming method allows pure functional languages like Haskell to, in a highly modular and abstract way, simulate effects that are found in imperative languages, for example global states, exception handling, non-determinism and input/output.

In pure functional languages, all flow of data is explicit which means that the value of an expression only depends on its free variables. Because of this it is comparatively easy to reason about the behaviour and correctness of functions in these languages, as it is always allowed to substitute equals for equals. This is called referential transparency.

But the explicitness of the data flow is also a burden. If one wants to add a feature (error handling, execution trace, etc) one has to rewrite every function in which the data generated from these features would flow through. Monads offer a way to hide these features, and also add the possibility to include new features without having to rewrite the functions of the program that perform the actual algorithms.

The main idea of a monad is that instead of having a function from `a -> b` you have a function from `a -> M b` where `M` is the (monadic) type constructor which hides the extra functionality.

[7]Actually, Liebnitz used the term monads for the spirits that were the fundamental units in his metaphysical description of the universe.

Two operations are needed to use the type `M`. The first, called unit, takes a value of type `a` and returns that value embedded in the monad.

```
return :: a -> M a
```

The second operation, `(>>=)`, often called bind, applies a function of type `a -> M b` to its first argument if that argument is a valid value of the type `M a`. Otherwise the orginal value is returned[8].

```
(>>=) :: M a -> (a -> M b) -> M b
```

An example of an value which is not valid is `Nothing` in the case the datatype `Maybe` is used as a monad. As `Nothing` has no value of type `a`, the function is not applied, and the original value, `Nothing`, is returned.

5.1 Use of monads in the thesis

Both the code for `string2struct` and the code for `struct2d` used in the parser generator in chapter 8 use monads. Haskell provides a class `Monad` with the functions `return` and `(>>=)`. This class is used to define the class `MonadZero` from which `MonadPlus` is defined. The class `MonadPlus` is used by `struct2d` and requires the following additional functions:

- `(++) :: MonadPlus a => a b -> a b -> a b`
- `zero :: MonadZero a => a b`

The function `zero` is needed as `MonadPlus` expands the class `MonadZero` and must have the functionality that `MonadZero` has.

This means that if one wishes to add some extra functionality to the parser generator one would not have to rewrite all the functions, all one would have to do is to create a monad with these functions.

In order to hide the explicit data value passing in the code, especially for functions taking two or more arguments, a number of monad combinators is used. These are defined in appendix D. The definition of the classes `Monad` and `MonadPlus` are in appendix C. For further information about monads, see [23, 5, 7].

6 Polytypic Programming

This section deals with one way of writing polytypic programs, using a syntactic construct specially designed for this. Other ways include using a universal datatype or using higher-order polymorphism and constructor classes (see [16, 12] for methods involving polymorphism and constructor classes). For current information about

[8] As the type of the value returned is changed, this implies that non-valid values must not contain any information about the type `a` if the expression should be type-correct.

polytypic programming research, see:

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

Many functions used in programming have to be used for several different types of data. In most typed programming languages, one is forced to write different instances of the function, one for each data type that the function is used on. Each instance of the function also has to have a separate name to distinguish it from the others.

One way of solving this is by overloading. But this only hides the fact that there are a number of instances. One still has to write every instance, and one cannot be sure that a specific instance exists without actually seeing the source code for that instance.

By generalizing a bit further, one can use polymorphic functions. These functions can take arbitrary data types as input but cannot themselves explicitly use the information in the input. One very common polymorphic function is `map`[9]:

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

The function `map` takes a function and a list and applies the function to each value in the list. Nothing about the values in the list is used by `map`, except the type of the values. If one would want to have a function that applies a function with the type `a -> b` on each element in a value of type `Tree a` (see above), a new function, call it mapTree, would have to be written. This function would behave very much like the map of lists, i.e. it would recursively traverse the value applying a function to each value in the tree.

Another way is to use polytypic functions. These are functions that are defined by induction over the structure of the user-defined datatype of the input to the function.

One limitation of polytypic functions are, as stated above, that they only take user-defined datatype with one free type variable. Another is that one cannot write polytypic functions that take values as input which have datatypes which include function spaces.

The parser generator that is described later in this paper uses polytypic functions. As an example of how polytypic functions are written, the polytypic function `pmap` as well as some common higher order polytypic functions are given. The function `pmap` is a generalization of `map` and is used in the main code for the parser in chapter 8. To write a polytypic function is to decide what action is needed depending on the *structure* of the datatype of the input. The datatype is modeled in PolyP as an initial functor-algebra. The idea behind using functor-algebras for this kind of

[9] Map has the type `Functor c => (a -> b) -> c a -> c b` in Haskell. But this is a form of overloading, as the Prelude of Haskell has instances for the ordinary list (written as [ ]) for the class Functor.

programming has been studied for many years (see [4, 17, 18]).

The seven components that are sufficient to describe the structure of a user-defined datatype will be explained by giving examples of datatypes that have these structures. For each datatype, its functor will be given. A functor consists of two parts: a mapping from a type to a type and a mapping from a function to a function.

6.1 The datatype Maybe

Consider the datatype `Maybe a`. This is a datatype which can be used as the answer of a computation that can either succeed or fail. In the case of a failure, the value `Nothing` is returned; otherwise the value `Just x` is returned, where `x` is the answer.

```
data Maybe a = Nothing | Just a
```

To analyse the datatype another datatype is constructed, called a functor, which describes the structure of `Maybe`. This datatype is called `FMaybe` and `Maybe` can be seen as it's fixpoint. As the only interest presently is in the structure of datatype, the names of the constructors `Nothing` and `Just` can be removed. In the case of nullary constructors such as `Nothing`, `()` is written. The infix symbol `|` is replaced with the prefix datatype name `Either` (which takes two types as arguments). As the datatypes dealt with only have one free type `a`, this can be kept. This gives the following declaration of `FMaybe`:

```
type FMaybe a b = Either () a
```

The `Either` datatype allows two values of different types to be used in `FMaybe`. To create these, two constructors are provided by Haskell:

```
Left ::  a -> Either a b
Right ::  b -> Either a b
```

To be able to transform values from `Maybe a` to `FMaybe a` and back, two more functions are needed:

```
inn ::  Fmaybe a -> Maybe a
out ::  Maybe a -> Fmaybe a
```

The two functions above can automatically be generated by PolyP if the datatype is a regular datatype. A datatype `D a` is regular if the argument of the datatype constructor on the left- and right-hand side are the same in its definition and it contains no functions spaces (for a discussion on allowing functions spaces, see [19]).

Using `Maybe` as an example, the instances of `inn` and `out` produced by PolyP would be:

```
inn x = either (uncurry0 Nothing) (uncurry1 Just)
out x = case x of
          Nothing -> Left ()
          Just a  -> Right a
```

The `either` function takes three arguments of which the two first are functions. The third argument is of the type `Either a b`. If the actual value stored in the third argument is of type `a` the first function is applied to it, otherwise the second function. The functions `uncurry0` and `uncurry1` and produced by PolyP to handle the data flow.

We now have one part of the functor that is needed for `pmap`. `FMaybe` is a mapping from a type to a type but to fully describe the functor for `Maybe`, we need also need a mapping from a function to a function. This part is called `fmap`. The most basic `fmap` is the identify function, `id`. Another very basic function is the `fmap` on sums (represented by the `Either` datatype) called `(-+-)`:

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

With these `fmap` functions we can describe the `fmap` for `FMaybe a`.

```
fmap ::  (a -> b) -> (c -> d) -> FMaybe a c -> FMaybe b d
fmap = \f g -> id -+- f
```
[10]

The instance of `map` from `Maybe a` which uses `fmap` is called `pmap` (for polytypic map).

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

`Regular` is used in PolyP to ensure that the user-defined datatype is regular, as described above.

To describe the structure of datatypes, PolyP uses a special notation in polytypic functions. Values without information (as `()`) are written as `Empty`. A value with information (`Just a`) are written as `Par`. Sums are specified by using the infix symbol `+`.

[10] The reason for the presently rather useless parameter g will be explained in section 6.2, below.

6.2 The datatype List

Now consider the datatype `List a`. Using the method for `Maybe` to derive `FMaybe`, the corresponding `FList` is constructed. `List a`, however, has a recursive element in the constructor `Cons`. The recursive value can be kept as it is, as PolyP only analyses the structure of a datatype one level at a time[11]. `Cons` has two parts, it consists of a pair which is represented by the ordinary pair constructor `(,)`. In the functor-algebra used in PolyP, `(,)` has higher precedence that `Either` and the datatype `Either` is only allowed at the top level of expressions.

```
data List a = Nil | Cons a (List a)
type FList a b = Either () (a,b)
```

The use of pairs for the juxtaposition requires the definition of a fmap for pairs, called `-*-`.

```
(-*-) ::  (a -> c) -> (b -> d) -> (a,b) -> (c,d)
(f -*- g) (x,y) = (f x,g y)
```

This definition allows us to define `fmap` and `pmap` for `Flist` and `List` respectively.

```
fmap ::  (a -> b) -> (c -> d) -> FList a c -> FList b d
fmap = \f g -> id -+- (f -*- g)
pmap ::  Regular List => (a -> b) -> List a -> List b
pmap f = inn .  fmap f (pmap f) .  out
```

Note that the definition of `pmap` is exactly the same as for `Maybe`. The reason `fmap` for `Maybe` has two arguments is that normally a functor has a recursive element and the second argument to `fmap` takes care of the recursive case.

The presence of a recursive value is written as `Rec` in polytypic functions in PolyP. In the same way, a juxtaposition is written as `*`.

6.3 The datatype Tree

The datatype `Tree a` is similar in structure to `List a` which is reflected in how the functor `FTree` is represented and the definition of `fmap`. Note that `pmap` is exactly the same function as defined on `Maybe a` and `List a`.

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
type Ftree a b = Either a (b,b)
```

```
fmap ::  (a -> b) -> (c -> d) -> FTree a c -> FTree b d
fmap = \f g -> f -+- (g -*- g)
```

[11] I.e. it analyses recursive values by applying a new instance of itself to the recursive value.

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

6.4 The datatype Rose

There are only two cases in the type structure left to be dealt with. One is the trivial case, `Const t`, which deals with constant datatypes in the user-defined datatypes. The other is when a constructor uses another datatype. This is exemplified by the earlier mentioned `Rose a`. A rose tree consists of a pair of a value and a list of rose trees. This list is also a user-defined datatype.

```
data Rose a = Fork a (List (Rose a))
type FRose a b = (a, List b)

fmap ::  (a -> b) -> (c -> d) -> FRose a c -> FRose b d
fmap = \f g -> f -*- pmap (fmap f g)
pmap ::  Regular Rose => (a -> b) -> Rose a -> Rose b
pmap f = inn .  fmap f (pmap f) .  out
```

The `pmap` which is used inside `fmap` for `FRose` is *not* the `FRose` version. So this declaration of `fmap` relies on the fact that there is another instance of `pmap` for the datatype `List`. This is not a problem in PolyP, as PolyP can create every instance of `fmap` that is needed from the polytypic declaration of `fmap`.

The use of other datatypes within a datatype is notated by the infix symbol `@` in PolyP.

6.5 The polytypic map → pmap

Given the examples above, we now know what `fmap` should do for each kind of component that describes a regular datatype. Using the notation described in the examples above, all possible structures of the input datatype can be described. The complete code of `fmap` and `pmap` in PolyP is as follows:

```
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)
            Empty     ->  id
            Par       ->  p
            Rec       ->  r
            d @ g     ->  pmap (fmap p r)
            Const t   ->  id

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

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

6.6 Some polytypic functions

Some very useful higher-order polytypic functions are easily written. One such function is the catamorphism, which is the polytypic equivalence of `foldr` on lists. It takes a function of the type `f a b -> b` and a value and then recursively replaces the constructors from its second argument with the functions provided by the first argument.

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

`FunctorOf` is used to specify PolyP's internal representation of a type.
The anamorphism is the inverse of a catamorphism in the sense that it constructs a value of a type from something else.

```
ana  :: Regular d => (b -> FunctorOf d a b) -> (b -> d a)
ana    o = inn . fmap id (ana    o) . o
```

The hylomorphism is the composition of an anamorphism and a catamorphism. It can be used when it is advantageous to have an intermediate data type for calculations in a problem.

```
hylo :: Bifunctor f => (f a b -> d) -> (c -> f a b) -> c -> d
hylo i o = i    . fmap id (hylo i o) . o
```

Another useful polytypic function which is very easy to write is `constructors`. It is actually a constant which is a list of all constructors for a user-defined datatype, but as PolyP decides which type it has depending on where it is used, it can be seen as a kind of function. In each place where a constructor needs an argument, these arguments are left undefined.

```
constructors  :: [d a]
constructors  =  map inn fconstructors

polytypic fconstructors :: [f a b]
 = case f of
     g + h     ->  map Left fconstructors ++ map Right fconstructors
     g         ->  [undefined]
```

`Thread` is used to lift monads. The function takes a regular datatype with a monadic value and transforms it to a monad of the regular datatype.

```
thread :: (Regular d, Monad m) => d (m a) -> m (d a)
thread  =  cata (mapm inn . fthread)

polytypic fthread :: Bifunctor f => f (m a) (m b) -> m (f a b)
  = case f of
      g + h     ->  sumthread  . (fthread -+- fthread)
      g * h     ->  prodthread . (fthread -*- fthread)
      Empty     ->  return
      Par       ->  id
      Rec       ->  id
      d @ g     ->  thread     . (pmap fthread)
      Const t   ->  return

sumthread  :: Either (m a) (m b) -> m (Either a b)
sumthread  =  mapm Left `either` mapm Right

prodthread :: (m a,m b) -> m (a,b)
prodthread (mx,my) = mx >>= \x -> my >>= \y -> return (x,y)

mapm :: Monad m => (a -> b) -> m a -> m b
mapm f mx = mx >>= \x -> return (f x)
```

For a more complete introduction to polytypic programming, see [13, 15].

7 Locally Quantified Datatypes

A locally quantified datatype hides the types of values used in user-defined datatypes. One example of such a datatype is `Secret`:

```
data Secret = Secret ?a
```

Here the symbol `?` specifies that `a` is a locally quantified datatype. When the constructor `Secret` is applied to any value the resulting type is simply `Secret`. Two `Secret` values created from different datatypes are treated as being of the same datatype. But to ensure type-correctness in a program, the quantified datatype must not be allowed to escape from its local state. This means that the value in the datatype cannot be used as an argument to a function which would return a value of a type which would expose the type of the locally quantified datatype.

The user-defined datatype `Value a` which is used in the parser generator uses a locally quantified datatype `?d (Struct a)` for its `Comp` constructor. The `Comp` value stores three values: The first is a list of the constructors used to build values of the locally quantified datatype. The second is a pretty printer for the type `d (Struct`

a) and the last argument is a `Struct` which was produced when pretty printing a value with the second argument.

By applying the second argument to the first one, the datatype does not escape and it is possible to store different datatypes in different values constructed by using `Comp`. Without this method, it would be impossible to generate a type-correct parser for datatypes which are defined in terms of other user-defined datatypes.

Locally quantified datatypes are supported by the Chalmers Haskell Compiler [1] which checks at compile-time that these datatypes do not escape their local state.

8 A parser generator using PolyP

The main point of this paper is to show how to construct a parser generator from the methods described above. After some experiments, it was found that the most practical way of creating a parser generator with PolyP was to introduce an intermediate datatype named `Struct`. This meant that the pretty printer would not give a `String` as output but rather a value of the type `Struct`. This actually adds modularity to the pretty printer. When a `Struct` value has been created by the pretty printer one can give that value to different "back-ends" that can produce output in a number of different ways: a `String`, formatted text like LaTex or HTML or as a picture. Because of this intermediate stage, a total of four major functions are needed[12]:

```
d2struct :: d a -> Struct a
struct2string :: Struct a -> String
string2struct :: (Parser String a) -> Parser String (Struct a)
struct2d :: (d a -> Struct a) -> [Struct a] -> Maybe (d a)
```

This means that `string2struct` can parse a `String` to a `Struct a` and `struct2d` can then transform it into the datatype `d a`. By using monad operators these two functions can together form a composite function that receives a parser for the type `a` and a prettyprinter for the datatype `d a`, and return a parser with lexical analyser included.

8.1 Pretty printers

Writing pretty printers for the datatypes `List` and `Tree` used in the parser generator is straightforward. The only questions that might arise concern in which order values in a juxtaposition will be printed. The values may be in any order as long as the first value of the `Arg` constructor is the same as the data values position in the juxtaposition. A limitation of the pretty printers that can be used in the context of this paper is that a pretty printer for a certain datatype may not apply another pretty printer to any value of that datatype[13].

[12] Actually, two more arguments are needed for string2struct, which will be show in section 8.3.

[13] See the chapter 9, Conclusions on page 26 for a discussion on this.

Examples of pretty printers:

```
pp_L :: List a -> Struct a
pp_L x = case x of
  Nil          ->  Symb "Nil"
  (Cons a as)  ->  Beside (Symb "Cons")
                          (Beside (Arg 1 (Par a))
                                  (Arg 2 (Rec (pp_L as))))

pp_T :: Tree a -> Struct a
pp_T x = case x of
  (Leaf a)    ->  Beside (Symb "Leaf")
                         (Arg 1 (Par a)
  (Node l r)  ->  Beside (Symb "Node")
                         (Beside (Arg 1 (Rec (pp_T l)))
                                 (Arg 2 (Rec (pp_T r))))
```

The values stored in a datatype do not have to be pretty printed in the same order as they appear in the constructor. For example, the pretty printer `pp_T'` takes a value of type `Tree a` but prints the value of the right subtree before the left subtree. Note that the first argument of the constructor `Arg` matches the second argument's place in the value being pretty printed.

```
pp_T :: Tree a -> Struct a
pp_T x = case x of
  (Leaf a)    ->  Beside (Symb "Leaf")
                         (Arg 1 (Par a)
  (Node l r)  ->  Beside (Symb "Node")
                         (Beside (Arg 2 (Rec (pp_T r)))
                                 (Arg 1 (Rec (pp_T l))))
```

In the function `pp_R`, which is a pretty printer for `Rose a`, the `Comp` constructor needs a list of `?d (Struct a)`. This is produced by taking the list of type `?d a` and mapping the `Arg` constructor to each value of the type `a`. As the type `?d a` is unknown, the polytypic function `pmap` is used to apply the constructor to each value.

```
pp_R :: Rose a -> Struct a
pp_R (Fork a as) =
  Beside (Symb "Fork")
    (Beside (Arg 1 (Par a))
       (Arg 2 (Comp
                 (map (pmap ((Arg 1) . Par)) constructors)
                 pp_L
                 (pp_L (pmap pp_R as)))))
```

This pretty printer uses the pretty printer for the datatype `List`, as the datatype contains values of the type `List (Rose a)`. It also uses the polytypic function `pmap` (from page 16) which means that `pp_R` does not have to be rewritten if the definition of `List` changes.

8.2 struct2string

The function `struct2string` is one of the possible "back-ends" for the pretty printer and is trivial to write. It takes an argument of type `Struct a` and transforms it into a `String`. More complex versions can be written, but the general printer is not an actual part of the problem of constructing a parser generator. The code for it is given on page 8 as an example of a pretty printer.

8.3 string2struct

Using methods from Section 4, it is possible to construct a parser that parses a `String` to the `Struct a` value that would be its representation if one parsed it to the type `d a` and then applied a pretty printer. Three things are needed:

- A list of the constructors of the datatype. For example, the datatype `List a` has the following list of constructors:

  ```
  constructors_L = [Nil,Cons undefined undefined]
  ```

  This list, or a similar list for any datatype can be constructed by a simple polytypic function `constructors`, see page 17. This is the only part of `string2struct` that uses polytypic programming.

- A pretty printer that has the type `d a -> Struct a`. If mapped on the list of constructors a list of `Struct` is returned. In the case of `List a` and the pretty printer `pp_L`, the result is:

  ```
  [Symb "Nil", Beside
                 (Symb "Cons")
                 (Beside
                    (Arg 1 (Par undefined))
                    (Arg 2 (Rec undefined)))]
  ```

- A parser for the type `a`. This is used for parsing the occurences of `a` in the input. For example, the function `number` (see page 10) can be used if `a` is `Int`.

We also want to check that parentheses are used in a legal way. `string2struct` takes the constructor list, the pretty printer and the parser and creates one parser for each possible constructor that can be parsed. `try_Parse` is then used to apply the input to each of these parsers.

```
string2struct :: [d a] -> (d a -> Struct a) ->
                 Parser String a -> Parser String (Struct a)
string2struct cl pp parse_par s =
  try_Parse (map (parse_constructor cl pp parse_par . pp) cl)

try_Parse :: [Parser String (Struct a)] -> Parser String (Struct a)
try_Parse = foldr (+^+) failure
```

Finally, `parse_constructor` checks that a `Struct` value can correspond to the head of the string. It does this by knowing how the different values of `Struct` and `Value` are printed by the general printer. In the case of a `Symb` value, the `String` should start with exactly the characters of `Symb`. The `Beside` value indicates that the function should first parse the left hand side and then the right hand side. The `Arg` value requires the use of `parse_args` to parse the `Value` stored in `Arg`.

The function `parse_args` is similar. In the case of a `Rec` value, the string to be parsed should be something `string2struct` could parse. In the case of a `Par` value, the `String` should be something that could be parsed by the parser for type `a`.

The function `par` is used to allow parentheses around values[14].

```
parse_constructor :: [d a] ->
                 (d a -> Struct a) ->
                 (Parser String a) ->
                 Struct a ->
                 Parser String (Struct a)
parse_constructor cl pp parse_par = par . k
-- k :: Struct a -> Parser String (Struct a)
  where k (Symb c)     = text (lexical c) `into`
                         \_ -> succeed (Symb c)
        k (Arg n x)    = par (parse_args cl pp parse_par x) `into`
                         succeed . (Arg n)
        k (Beside x y) = k x `into` \v  ->
                         k y `into`
                         succeed . (Beside v)

parse_args :: [d a] ->
                 (d a -> Struct a) ->
                 (Parser String a) ->
                 Value a ->
                 Parser String (Value a)
parse_args cl pp parse_par = k
-- k :: Value a -> Parser String (Value a)
```

[14] This system allows some expressions *without* parentheses which are not allowed in Haskell. For example, `Cons 2 Cons 5 Nil)` can be parsed by the system as a value of `Num a => List a` but this is not type-correct in Haskell as the outer `Cons` actually receives four arguments.

```
  where k (Par a)          = succeed Par `ap` parse_par
        k (Rec a)          = succeed Rec `ap` (s2s cl pp parse_par)
        k (Comp cl' pp' x) = succeed (Comp cl' pp') `ap`
                               (s2s cl' pp' (s2s cl pp parse_par))

par :: Parser String a -> Parser String a
par p = p `alt` (literal "(" `into` \_ ->
                 p `into` \x ->
                 literal ")" `into` \_ ->
                 succeed x)
```

8.4 Struct2d

This function is the part of the parser generator which heavily depends on PolyP's ability to handle polytypic functions.

The function `struct2d` takes a list of `Struct a` which is lazily produced by pretty printing all the constructors of the datatype. Each of these is then compared with the `Struct a` which was produced by `string2struct`. The first one that matches[15] the structure of the `Struct` value is used to construct the actual value. Note that the function only checks the top level as the lower levels are not yet created (they are left undefined).

```
struct2d :: (d a -> Struct a) -> Parser (Struct a) (d a)
struct2d pp = s2d (map (beside2list . pp) constructors) . beside2list

s2d :: (MonadPlus m, Poly (FunctorOf d)) =>
            [[Struct a]] -> [Struct a] -> m (d a)
s2d cl = s2f cl >>>
           (mapm inn . fthread . fmap return (s2d cl . beside2list))
```

To make the comparison easier, each `Struct a` is converted into a list of `Struct a` by `beside2list` where all `Beside` values have been removed.

```
beside2list :: Struct a -> [Struct a]
beside2list f = case f of
                (Beside l r) -> beside2list l ++ beside2list r
                f            -> [f]
```

The function `s2f` tries each of the lists of the type `Struct` which was produced from the constructors against the lists produced from the `Struct` from `string2struct`. This must be done in a polytypic way as datatypes can have many constructors and `s2f` must match each `Struct` value with its correct constructor. PolyP uses `Left`

[15] It will always match one of the constructors as this is tested in `string2struct`, but `struct2d` has to check which of the constructors match.

and Right to mark which constructor of the original datatype is used and if it is applied correctly to the value constructed.

```
polytypic s2f  :: MonadPlus m => [[Struct a]] -> [Struct a] ->
                  m (f a (Struct a))
= case f of
    f + g -> (fixpar . head >>>> (mapm (Left . fst) . s2Value))
             +^^+ (s2f . tail >>>> (return . Right))
    f     -> fixpar . head >>>> (mapm fst . s2Value)
```

The lists of `Struct a` are compared in `fixpar` by zipping them together and then applying the result first to the function `removesymbols`. This removes all instances of `Symb` as these carry no information[16] and checks that the `Arg` elements are in the same order in both lists. If this is not true `fixpar` returns `Nothing`. Otherwise `fixpar` gives the second list to `sortargs` which quicksorts it according to the numerical value of the `Arg` elements (so the first argument is first etc) and only keeps the values of type `Value` stored in `Arg`, producing a list of the type `Value`. This is done to ensure that the values are in the order required by the constructor.

```
fixpar :: MonadZero m => [Struct a] -> [Struct b] -> m [Value b]
fixpar xl = mapm stripsort . removesymbols . (zip xl)

removesymbols :: MonadZero m => [(Struct a,Struct b)] -> m [Struct b]
removesymbols xl = case xl of
  []                      -> return []
  ((Symb a,Symb b):xl)    -> if a == b then removesymbols xl
                             else zero
  ((Arg n x,Arg m y):xl)  -> if n == m
                             then mapm (\ys->(Arg m y):ys)
                                       (removesymbols xs)
                             else zero
  _                       -> zero

stripsort :: [Struct a] -> [Value a]
stripsort = map stripArg . sortargs

stripArg :: Struct a -> Value a
stripArg (Arg _ x) = x

sortargs :: [Struct a] -> [Struct a]
sortargs xl = case xl of
  []              -> []
  ((Arg n x):xl)  -> let lhs = (filter (\(Arg m _) -> m<=n) xl)
```

[16] It actually checks that the `Symb` values are equal as an extra safe-guard.

```
                        rhs = (filter (\(Arg m _) -> n<=m) xl)
                    in lhs ++ [Arg n x] ++ rhs
```

The function s2Value checks that at each place where a value is needed to construct the final value the correct constructor from Value is used. This is needed to ensure the type correctness of the parsed value. If one or more of these values are incorrect, the whole matching fails and the function fails.

```
s2Value :: :: MonadZero m => [Value a] -> m (f a (Struct a),[Value a])
polytypic s2Value = case f of
  f * g   -> s2Value >>*  s2Value
  Empty   -> return . pair ()
  Par     -> s2Par >>: \x -> (return . pair x)
  Rec     -> s2Rec >>: \x -> (return . pair x)
  d @ g   -> s2Comp >>: \x -> (return . pair x)

pair x y = (x,y)

s2Par x = case x of
  Par y -> return y
  _     -> zero

s2Rec x = case x of
  Rec y -> return y
  _     -> zero

s2Comp x = case x of
  (Comp cl pp y) -> s2d (map (beside2list . pp) cl)
                        (beside2list y)
  _              -> zero
```

8.5 The parser

The function parser finally constructs a parser when given a pretty printer[17] and a parser for the datatype a.

```
parser :: (d a -> Struct a) -> Parser String a ->
          Parser String (d a)
parser pp parse_par = string2struct constructors pp parse_par >>>
                      (struct2d pp . beside2list *>> return)
```

[17]By looking at the code, one can see that it is actually sufficient to give a list of the partially pretty printed constructors instead of a pretty printer. So why not give this instead?
First, one will almost always anyway want a pretty printer, and secondly, if one changes the representation of the pretty printer the parser changes also. If one uses a list, one will have to rewrite it each time the representation is changed.

9 Conclusion

The parser generator described here produces deterministic recursive-descent parsers for LL(1) grammars. These grammars are represented by pretty printers. However, the generator does not check for left recursion or ambiguity in the grammar. This means that the responsibility of not having an ambiguous grammar falls on the writer of the pretty printer.

Infinite loops due to left recursion can occur with parsers generated with this system. This is avoided by not using pretty printers that have a recursive call at the start of its output. A more subtle version of left recursion occures when using `Comp` and for the "inner" datatype have a pretty printer that first prints the `Par` value.

The parser generated here can only take a pretty printer that produces exactly the same type of output for each recursive step. It does not allow pretty printers which print lists on the standard form [1,2,3] as the first element has a '[' symbol which does not appear anywhere else. This is due to the fact that each recursive value that is being parsed is treated as an original value. The reason for this is that polytypic functions only analyse the argument one level at the time and the present parser generator only generates one level of the datatype it is trying to parse. If one write a pretty printer like this and generate a parser from it, the parser will never use the information from the functions to pretty print the recursive step. In the case of the pretty printer mentioned above, the parser generated would be able to parse expression on the form [1[2[3].

Even though the generated parsers have these limitations it is the hope of this paper to show that it is not only possible but with further research this method can in time offer a substantial reduction in time and resource when constructing parsers. The goal would be to have the polytypic programming totally integrated with Haskell so that parsers could be produced on the fly without the programmer having to think about it.

10 Future Work

One can easily change the parser generator to handle ambiguous grammars by changing `Maybe` to a list in the type `Return`. This would make it possible to test it on more natural languages.

There has been no experiment of trying to write a pretty printer for a larger datatype, for example for an abstract syntax tree of a small language. This would give a better comparison between the difficulty of writing the pretty printer used by the system and writing the parser manually. It would also make it possible to make some measurements of the effectivity of the generated parser in terms of speed and memory usage.

Another thing one might do is to rewrite the parser to use Continuation based parsing as in [22]. If one uses continuation based pretty printers, one could catch these continuations to create parsers that can handle the problem with unique starting sequences, as was the case with list in the section above. This would actually be a (probably very ineffective) code-cracker which, when given a function that codes a message can try different keys to crack the code.

References

[1] L. Augustsson. HBC – The Chalmers Haskell compiler. More information (and the code) is available from `http://www.cs.chalmers.se/~augustss/hbc/hbc.html`.

[2] Cup: LALR Parser Generator for Java. See `http://www.cc.gatech.edu/gvu/people/Faculty/hudson/java_cup/home.html`

[3] The Derive homepage: `http://www.dcs.gla.ac.uk/~nww/derivehome.html`.

[4] T. Hagino. *Category Theoretic Approach to Data Types.* PhD thesis, University of Edinburgh, 1987.

[5] G. Hutton and E. Meijer. Monadic Parser Combinators, Appears as technical report NOTTCS-TR-96-4.

[6] The GNU Project: `http://www.gnu.ai.mit.edu/home.html`.

[7] J. Hughes and M. Carlsson. Systematic Design of Monads. Available to download from `http://www.cs.chalmers.se/~rjmh/AFP/monads.ps`.

[8] J. Hughes. The Design of a Pretty-printing Library. In J. Jeuring and E. Meijer, editors *Advanced Functional Programming* LNCS 925, pages 53-96, Springer-Verlag, 1995. Available to Download from `http://www.cs.chalmers.se/~rjmh/Papers/pretty.ps`.

[9] M. Huisman. The calculation of a polytypic parser. Master thesis, Utrecht University, Inf/Src-96-19.

[10] The Java Compiler Compiler: `http://www.suntest.com/Jack`.

[11] P. Jansson: a library of polytypic functions. see `http://www.cs.chalmers.se/~patrikj/poly/polyp/polylib/`.

[12] P. Jansson and J. Jeuring. Polytypic unification – implementing polytypic functions with constructor classes. Submitted for publication 1996, see `http://www.cs.chalmers.se/~johanj/polytypism`.

[13] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *Principles of Programming Languages, POPL '97*, pages 470-482, 1997.

[14] Jell: Parser Generator. Information can be found at `http://www.sbktech.org/jell.html`.

[15] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer and T. Sheard *Advanced Functional Programming*, LNCS 1129, pages 68–114, Springer-Verlag, 1996.

[16] M 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.

[17] D.J. Lehmann and M.B. Smyth. Algebraic specification of data types: A synthetic approach. *Math. Systems Theory* 14:07-139. 1981.

[18] E.G. Manes and M.A. Arbib. *Algebraic Approaches to Program Semantics*. Text and Monographs in Computer Science. Springer-Verlag, 1986.

[19] E. Meijer and G. Hutton. Bananas in space: Extending fold and unfold to exposition types. In *Conference Record of FPCA '95, SIGPLAN-SIGARCH-WC2.8 Conference of Functional Programming Languages and Computer Architecture*, pages 324-333, 1995.

[20] E. Moggi, Computational lambda-calculus and monads. In *Symposium on Logic in Computer Science*, Asilomar, California, IEEE, June 1989. (A longer version is available as a technical report from the University of Edinburgh.)

[21] E. Moggi, An abstract view of programming languages. Course notes, Univerisity of Edinburgh.

[22] Niklas Röjemo, Efficient parsing combinators. Available to download from `ftp://ftp.cs.chalmers.se/pub/users/rojemo/thesis/part4.ps.Z`.

[23] P. Wadler. Monads for functional programming, appears in *Advanced Functional Programming*, Proceedings of the Båstad Spring School, pages 24-52, May 1995, Springer Verlag Lecture Notes in Computer Science 925. Available to download from
`http://www.dcs.gla.ac.uk/ wadler/papers/marktoberdorf/marktoberdorf.ps.gz`.

[24] P. Wadler, Comprehending monads. In *Mathematical Structures in Computer Science*, volume 2, pages 461-493, 1992.

[25] P. Wadler, The essence of functional programming (invited talk). In *19:th Symposium on Principles of Programming Languages*, pages 1-14, Albuquerque, New Mexico, ACM, January 1992.

A The Program

The program is split into a number of files. These files are grouped into two categories: The ones which have to be compiled with PolyP and those that have to be compiled with hbc. The PolyP files can be compiled separately from the other ones and they can be tested in Hugs (the Haskell User's Gofer System).

The PolyP files are:

- Base.phs: Library file with the polytypic function `fmap`.
- ConstructorName.phs: Library file with the polytypic function `constructors`.
- Thread.phs: Library file with the polytypic function thread.
- Main.phs: The `s2d` function with definitions of `Value`, `Struct` and pretty printers that work in PolyP. It includes the main function which makes PolyP produce two versions of struct2d, `struct2d_f4List` and struct2d_f4Rose for parsing `Struct` values to the datatypes `List a` and `Rose a`.
- expltypes.hs: This file has the type definitions of the necessary functions produced by PolyP. They are needed to avoid unresolved top-level overloading in Hugs.

The Library files comes from Patrik Janson's library of polytypic functions [11].

The hbc files are:

- Monad.hs: Definitions of the monadic operators used to hide value passing.
- Datatypes.hs: The definitions of `Value` and `Struct` using locally quantified datatypes.
- Parser.hs: The parser functions (as given in chapter 4 and appendix B).
- S2s.hs: The Parser for `Struct a` (as given in section 8.3).
- Example_pretty_printers.hs: Examples of pretty printers.
- Test.hs: A slightly modified version of the code produced by PolyP from Main.phs.
- Main.hs: The main file for compiling the system with hbc. Contains a main function which tries to parse input from standard input and shows the result on standard output.

To compile the PolyP files the following command is used (this assumes that all files are in the present directory and that PolyP has been installed):

```
cat Base.phs ConstructorName.phs Thread.phs Main.phs >
temp.phs && polyp temp.phs | cat expltypes.hs - > test.hs
```

Some modifiction must unfortunately be done manually to the test.hs file to be able to use it in Hugs. Most of these changes are because PolyP can not handle locally quantified datatypes or because functions are missing from the prelude. The following changes are needed:

- One instance of pp_R should be removed.
- The following functions declarations must be removed (they are just dummies):
  - s2d'
  - filter
  - zero
  - (++-)
- The application of ++- should be replaced by ++.
- The text

```
(Comp cl pp x) -> s2d' (map (beside2list .pp) cl)
                        (beside2list x)
```

  must be replaced by

```
(Comp cl pp x) -> s2d_f4List
                    (map (beside2list . pp_L) constructors_f4List)
                    (beside2list x)
```

  If one is only going to use the code to compile the program in hbc [18], only the word s2d' has to be changed to s2d_f4List.

To be able to use test.hs as a part of program compiled by hbc, the following modifications need to be done:

- Add to following to the beginning of the file:

```
module Test where

import Datatypes
import Monad
import Example_pretty_printers
```

- Remove all datatype definitions (List, Rose, Tree, Struct and Value).
- Remove pp_L,pp_R and pp_T. These are defined in Example_pretty_printers.hs and use another version of the datatypes Struct and Value.
- Remove ++, ++, ++, >>>, >>>>, >>:, >>* and pair.
- Remove main (and its type declaration) as Main.hs has another definition of main in it.

As before, most of these changes are due to the fact that PolyP cannot handle locally quantified datatypes (so that two versions of all datatypes and the functions which use them are needed).

[18] I.e. not use the code in Hugs.

B Parser combinators

The definitions of the functions given in chapter 4. The functions are based on the parsing combinators by Röjemo [22] and have only been modified to use the Monadic classes supported by Haskell.

```
type Return a b = Maybe (b,a)
type Parser a b = [a] -> Return b [a]

lexical :: String -> [String]
lexical i = lex' i
  where
    lex' [] = []
    lex' xxs@('\n':xs) = lex' xs
    lex' xxs@(x:xs) =
            if isSpace x then lex' xs
       else if isDigit x then token (span isDigit xxs)
       else if isAlpha x then token (span isAlpha xxs)
       else [x] : lex' xs
    token (t,xs) = t : lex' xs

failure :: Parser a b
failure input = zero

succeed :: b -> Parser a b
succeed a input = Just (a,input)

satisfy :: (a -> Bool) -> Parser a a
satisfy p (x:xs) =
     if p x
     then return (x,xs)
     else zero
satisfy p [] = zero

into :: Parser a b -> (b -> Parser a c) -> Parser a c
into p1 p2 input = p1 input >>= \(v,input') -> p2 v input'

alt :: (Parser a b) -> (Parser a b) -> (Parser a b)
alt p1 p2 input = (p1 input) ++ (p2 input)

ap :: Parser a (b->c) -> Parser a b -> Parser a c
p1 `ap` p2 = p1 `into` \f -> p2 `into` \x -> succeed (f x)
```

C Functors and Monadic operators in Haskell

These are taken from the standard Prelude used in Hugs.

```
class Functor f where
   map :: (a -> b) -> (f a -> f b)

class Monad m where
   (>>=)  :: m a -> (a -> m b) -> m b
   return :: a -> m a

   (>>)   :: m a -> m b -> m b
   p >> q  = do {p;q}

class Monad m => MonadZero m where
   zero   :: m a

class MonadZero m => MonadPlus m where
   (++)   :: m a -> m a -> m a

instance Functor Maybe where
   map f Nothing  = Nothing
   map f (Just x) = Just (f x)

instance Functor [] where
   map f []     = []
   map f (x:xs) = f x : map f xs

instance Monad Maybe where
   Just x  >>= k  =  k x
   Nothing >>= k  =  Nothing
   return         =  Just

instance MonadZero Maybe where
   zero           = Nothing

instance MonadPlus Maybe where
   Nothing ++ ys  = ys
   xs      ++ ys  = xs

instance Monad [ ] where
   (x:xs) >>= f  =  f x ++ (xs >>= f)
   []     >>= f  =  []
   return x      =  [x]
```

```
instance MonadZero [ ] where
   zero = []

instance MonadPlus [ ] where
   []     ++ ys = ys
   (x:xs) ++ ys = x : (xs ++ ys)
```

D Monadic Combinators

These are the monadic operators to hide the explicit value passing in the program.

```
(>>>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
(f >>> g) = \x -> (f x >>= g)

(>>>>) :: Monad m => (a -> b -> m c) -> (c -> m d) -> a -> b -> m d
(f >>>> g) = \x y -> (f x y >>= g)

(+^+) :: MonadPlus m => (a -> m b) -> (a -> m b) -> a -> m b
(f +^+ g) = \x -> (f x ++ g x)

(+^^+) :: MonadPlus m => (a -> b -> m c) -> (a -> b -> m c) -> a -> b -> m c
(f +^^+ g) = \x y -> (f x y ++ g x y)

(+^^^+) :: MonadPlus m => (a -> b -> c -> m d) ->
                          (a -> b -> c -> m d) ->
                           a -> b -> c -> m s
(f +^^^+ g) = \x y z -> (f x y z ++ g x y z)

(>>:) :: MonadZero m => (a -> m c) -> (c -> [a] -> m d) -> [a] -> m d
(f >>: g) = \x -> case x of
                []   -> zero
                (x:xl) -> f x >>= \y -> g y xl

(>>*) :: Monad m => (a -> m (c,d)) -> (d -> m (e,f)) -> a -> m ((c,e),f)
(f >>* g) = f >>> \(x,z) -> g z >>= \(y,w) -> return ((x,y),w)



(*>>) :: Monad m => (a -> m b) -> ((b,c) -> m f) -> (a,c) -> m f
(f *>> g) = \(x,y) -> f x >>= \z -> g (z,y)
```