A Functional-Logic Perspective of Parsing

Rafael Caballero, Francisco J. López-Fraguas [*]

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid, {rafa,fraguas}@sip.ucm.es

Abstract. Parsing has been a traditional workbench for showing the virtues of declarative programming. Both logic and functional programming claim the ability of writing parsers in a natural and concise way. We address here the task from a functional-logic perspective. By modelling parsers as non-deterministic functions we achieve a very natural manner of building parsers, which combines the nicest properties of the functional and logic approaches. In particular, we are able to easily define within our framework parsers in a style very close to that of functional programming parsers, but using simpler concepts. Moreover, we have moved beyond usual declarative approaches to parsers, since the functional-logic parsers presented here can be considered as truly data values. Thus, interesting properties of the represented grammar, such as ambiguity, can be easily checked in our purely declarative setting.

1 Introduction

The problem of syntax analysis or *parsing* has been one of the most thoroughly studied issues in computer science. Its wide range of applications, from compiler development to natural languages recognition, is enough to attract the attention of any programming approach. This has also been the case for logic programming (LP, in short) and functional programming (FP, in short), and the parsing problem constitutes in fact one of the favorite fields for exhibiting the virtues of declarative programming, looking for a straightforward way of representing parsers as proper components of the language. This has been achieved by considering *recursive descendent parsers*, usually represented by means of language mechanisms adapted to simulate grammar rules (e.g. BNF rules).

From the point of view of LP, there is a more or less standard approach [SS86] to the construction of parsers in LP, which is based on a specific representation for grammars, the so-called *Definite Clause Grammars (DCG's)*. *DCG*'s are not logic programs, although they are readily translated to them. With *DCG*'s, one can hide the details of handling the input string to be parsed, which is passed from parser to parser using the LP technique of *difference lists* [SS86]. Parsing in LP benefits from the expressive power of non-determinism, which handles almost effortlessly the non-deterministic essence of grammar specifications. In the case of ambiguous grammars this means that multiple solutions are automatically provided where possible.

The use of logical variables and unification are also useful in LP parsers. They ease the construction of output representations, which is carried out explicitly by using an input/output

[*] Work partially supported by the Spanish CICYT (project TIC98-0445-C03-02/97 "TREND") and the ESPRIT Working Group 22457 (CCL-II).

extra argument. Moreover, multiple modes of use are allowed, i.e. LP parsers can be regarded as *generators* as well as recognizers of sentences, and parsers for some context sensitive languages can be easily defined.

The main contribution of FP parsers are the so called *higher order* combinators [Hut92,Fok95]. In addition, the use of *monads* (see [Wad95]), specially in combination with the *do notation* [Lau93,HM97] gives a very appealing structure to the parsers.

Many efforts have been done in the last decade in order to integrate LP and FP into a single paradigm, *functional-logic programming* (FLP in short, see [Han94] for a survey). As any other paradigm, FLP should develop its own programming techniques and methodologies, but little has been done from this point of view. In this paper the problem of developing FLP parsers in a systematic way is addressed, trying to answer the question: can FLP contribute significantly by itself (not just mimicking LP or FP) to the task of writing parsers?

We will show how a suitable combination of LP and FP features leads to parser definitions as expressive as FP parsers, but based on simpler concepts. Moreover, we have moved beyond current FP and LP approaches to parsers, for the FLP parsers presented here can be considered as *truly data values*. Thus, interesting properties of the represented grammar, such as ambiguity, can be easily checked in our purely declarative setting.

We stick to a view of FLP whose core notion is that of *non-deterministic function*. A framework for such an approach is given in [GH+99], which is extended to cope with higher-order features in [GHR97], and polymorphic algebraic types in [AR97]. In the approach to HO of [GHR97] functions are seen from an *intensional* point of view, where partially applied functions behave as free data constructors.

The rest of the paper is organized as follows. In the next section we will briefly describe the specific functional-logic language we are going to use: $\mathcal{TOY}$. Section 3 examines the main characteristics of parsers in LP and FP, choosing the best features of each paradigm to define our model of FLP parsers. Section 4 is devoted to the definition of some basic parsers and parser combinators. These functions are the basic pieces we will use to build more complicated parsers, like the examples presented in Section 5. In Section 6 we show how the 'intensional' view of functions allows $\mathcal{TOY}$ programs to manipulate parsers as truly data values. In particular, a suitable *fold* function for parsers in $\mathcal{TOY}$ is defined, together with an application of such function that checks whether a grammar is ambiguous. Finally, Section 7 summarizes some conclusions.

2 A succinct description of $\mathcal{TOY}$

All the programs in the next sections are written in $\mathcal{TOY}$[LS99], a purely declarative functional-logic language with solid theoretical foundations, which can be found in [GH+99,GHR97,AR97]. We present here the subset of the language relevant to this work (see [CLS97] for a more complete description and a number of representative examples).

A $\mathcal{TOY}$ program consists of *datatype*, *type alias* and *infix operator* definitions, and rules for defining *functions*. Syntax is mostly borrowed from Haskell [HAS98], with the remarkable exception that variables begin with upper-case letters whereas constructor symbols use lower-case, as function symbols do. In particular, functions are *curried* and the usual conventions about associativity of application hold.

Datatype definitions like `data nat = zero | suc nat`, define new (possibly polymorphic) *constructed types* and determine a set of *data constructors* for each type. The set of all data constructor symbols will be noted as CS (CS^n for all constructors of arity n).

Types $\tau, \tau', \ldots$ can be constructed types, tuples $(\tau_1, \ldots, \tau_n)$, or functional types of the form $\tau \to \tau'$. As usual, $\to$ associates to the right. $\mathcal{TOY}$ provides predefined types such as `[A]` (the type of polymorphic lists, for which Prolog notation is used), `bool` (with constants `true` and `false`), `int`, `real` for integer and real numbers, or `char` (with constants `'a'`,`'b'`, ...). *Type alias* definitions like `type parser A = [A]` $\to$ `[A]` are also allowed. Type alias are simply abbreviations, but they are useful for writing more readable, self-documenting programs. Strings (for which we have the definition `type string = [char]`) can also be written with double quotes. For instance, `"sugar"` is the same as `['s','u','g','a','r']`.

The purpose of a $\mathcal{TOY}$ program is to define a set FS of functions. Each $f \in FS$ comes with a given *program arity* which expresses the number of arguments that must be given to f in order to make it reducible. We use FS^n for the set of function symbols with program arity n. Each $f \in FS^n$ has an associated principal type of the form $\tau_1 \to \ldots \to \tau_m \to \tau$ (where τ does not contain $\to$). Number m is called the *type arity* of f and well-typedness implies that $m \geq n$. As usual in functional programming, types are inferred and, optionally, can be declared in the program.

With the symbols in CS, FS, together with a set of variables $X, Y, \ldots$, we form more complex expressions. We distinguish two important syntactic domains: *expressions* and *patterns*. *Expressions* are of the form $e ::= X \mid c \mid f \mid (e_1, \ldots, e_n) \mid (e\ e')$, where $c \in CS$, $f \in FS$. As usual, application associates to the left and parentheses can be omitted accordingly. Therefore $e\ e_1 \ldots e_n$ is the same as $((\ldots((e\ e_1)\ e_2)\ldots)\,e_n)$. Of course expressions are assumed to be well-typed. *Patterns* are a special kind of expressions which can be understood as denoting data values, i.e. values not subject to further evaluation, in contrast with expressions, which can be possibly reduced by means of the rules of the program. They are defined by $t ::= X \mid (t_1, \ldots, t_n) \mid c\ t_1 \ldots t_n \mid f\ t_1 \ldots t_n$, where $c \in CS^m$, $n \leq m$, $f \in FS^m$, $n < m$. Notice that partial applications (i.e., application to less arguments than indicated by the arity) of c and f are allowed as patterns, which are then called *HO patterns*, because they have a functional type. Therefore function symbols, when partially applied, behave as data constructors. HO patterns can be manipulated as any other patterns; in particular, they can be used for matching or checked for equality. With this *intensional* point of view, functions become 'first-class citizens' in a stronger sense that in the case of 'classical' FP. This treatment of HO features is borrowed from [GHR93,GHR97] and will constitute an useful tool in Sect. 6.

Each function $f \in FS^n$ is defined by a set of conditional rules of the form

$$f\ t_1 \ldots t_n\ = e\ \Longleftarrow\ e_1 == e'_1, \ldots, e_k == e'_k$$

where $(t_1 \ldots t_n)$ forms a tuple of linear (i.e., with no repeated variable) *patterns*, and e, e_i, e'_i are *expressions*. No other conditions (except well-typedness) are imposed to function definitions. The notation `V@pat`, with `V` a variable name and `pat` a valid pattern, is allowed for the patterns t_i. It represents the so-called *as patterns*: every occurrence of `V` in the body or the conditions of the rule will be automatically replaced by `pat`.

Rules have a conditional reading: $f\ t_1 \ldots t_n$ can be reduced to e if all the conditions $e_1 == e'_1, \ldots, e_k == e'_k$ are satisfied. The condition part is omitted if $k = 0$. The symbol `==` stands for *strict equality*, which is the suitable notion (see e.g. [Han94]) for equality when non-strict functions are considered. With this notion a condition `e == e'` can be read as: `e` and `e'` can be reduced to the same pattern. When used in the condition of a rule, `==` is better understood as a constraint (if it is not satisfiable, the computation fails), but the language contemplates also another use of `==` as a function, returning the value `true` in the case described above, but `false` when a clash of constructors is detected while reducing both sides. As a syntactic facility, $\mathcal{TOY}$ allows repeating variables in the head of rules but in this case repetitions are removed by introducing new variables and strict equations in the condition of the rule. As an example, the rule `f X X = 0` would be transformed into `f X Y = 0` $\Longleftarrow$ `X == Y`.

In addition to `==`, $\mathcal{TOY}$ incorporates other predefined functions like the arithmetic functions `+`,`*`, ..., or the functions `if_then` and `if_then_else`, for which the more usual syntax `if _ then _` and `if _ then _ else _` is allowed. Symbols `==`,`+`,`*` are all examples of *infix operators*. New operators can be defined in $\mathcal{TOY}$ by means of *infix* declarations, like `infixr 50 ++` which introduces `++` (used for list concatenation, with standard definition) as a right associative operator with priority 50. Operators for data constructors must begin with ':', like in the declaration `infix 40 :=`. *Sections*, or partial applications of infix operators, like `(==3)` or `(3==)` are also allowed.

Predicates are seen in $\mathcal{TOY}$ as `true`-valued functions. In spite of that, *clausal notation* is allowed for predicates, according to the syntax $p\ t_1 ... t_n\ :-b_1, \ldots, b_m$ which is simply a syntactic sugar for the functional rule $p\ t_1 ... t_n = true \Longleftarrow b_1 == true, \ldots, b_m == true$.

A distinguishing feature of $\mathcal{TOY}$, heavily used throughout this paper, is that no confluence properties are required for programs, and therefore functions can be *non-deterministic*, i.e. return several values for given (even ground) arguments. For example, the rules `coin = 0` and `coin = 1` constitute a valid definition for the 0-ary non-deterministic function `coin`. A possible reduction of `coin` would lead to the value `0`, but there is another one giving the value `1`. The system will try the first rule first, but if backtracking is required by a later failure or by request of the user, the second one will be tried. Another way of introducing non-determinism in the definition of a function is by putting *extra* variables in the right side of the rules, like in `z_list = [0|L]`. Although in this case `z_list` reduces only to [0|L], the free variable `L` can be later on instantiated to any list. Therefore, any list of integers is a possible value of `z_list`.

Our language adopts the so called *call-time choice* semantics for non-deterministic functions. Call-time choice has the following intuitive meaning: given a function call *(f* $e_1 \ldots e_n$*)*, one chooses some fixed value for each of the e_i before applying the rules for *f*. As an example, if we consider the function `double X = X+X`, then the expression `(double coin)` can be reduced to `0` and `2`, but not to `1`. As it is shown in [GH+99], call-time choice is perfectly compatible with non-strict semantics and lazy evaluation, provided *sharing* is performed for all the occurrences of a variable in the right-hand side of a rule.

Computing in $\mathcal{TOY}$ means solving *goals*, which take the form $e_1 == e'_1, \ldots, e_k == e'_k$, giving as result a substitution for the variables in the goal making it true. Evaluation of expressions (required for solving the conditions) is done by a variant of lazy narrowing based on the so-called *demand driven strategy* (see [LLR93]). With respect to higher-order functions, a first order translation following [Gon93] is performed.

3 Our model of parsers

In declarative programming we aim at defining parsers whose definition denote the structure of the underlying grammar. Consider the following production, written in extended BNF syntax

```
<expr> ::= <term><plus_minus><expr> | <term>
```

In order to translate properly this rule into a declarative program, we must come out to some decisions:
(i) How to represent the alternative of parsers, denoted in the rule above by the symbol |.
(ii) How to represent a sequence of parsers like `<term><plus_minus><expr>`.
(iii) Moreover, the parser `expr` should not only recognize when a sentence belongs to the underlying formal language. It should also return a suitable representation of the parsed sentence. Hence we must take care of representations when deciding the final structure of parsers.

Before considering these questions in our setting, we will briefly overview the characteristics of parsers in the two main declarative paradigms. In FLP we could remain attached to either the FP or the LP point of view, but we will show how a careful combination of both perspectives leads to the same expressiveness with simpler parser definitions. In the following discussion the LP point of view is represented by pure Prolog, while functional parsers are those of Haskell ([HAS98]) as described in [Fok95,Wad95,HM97].

3.1 Review of LP and FP parsers

As common ground, both paradigms represent the sentence to be parsed as a list of terminals. This list is provided as an input parameter to the parser, which tries to recognize a prefix returning the non-consumed part. This part, the output sentence, is then supplied as input for the next parser connected in sequence. Now we summarize some of the main differences between both approaches with respect to the points (i), (ii) and (iii) mentioned above.

In **Logic Programming** all the values need to be arguments of some predicate. Thus the input and output sentences and the representation must be parameters of the parser predicate. In addition to that

(i) The non-deterministic nature of grammar specifications is easily handled in Prolog, just representing different alternatives through different rules for the same predicate. The built-in mechanism of Prolog will initally choose the first rule. If it fails, or more solutions are requested by the user, the next alternative is tried by means of *backtracking*.

(ii) Input and output lists are explicit arguments when writing parsers in sequence, as witnessed by the Prolog definition: `expr(I,O) :− term(I,O1), plus_minus(O1,O2), expr(O2,O).` This notation is rather tedious, and is avoided in Prolog systems by introducing a new formalism which conceals the passing of parameters, namely the DCG's. Using DCG's one can write the more appealing rule `expr ⟶ term, plus_minus, expr.`
(iii) The representation also needs to be an argument, usually used as an output value. However, in this case, it is not a problem but an advantage, as it permits defining explicitly the construction of new representations from those of their components, as in the definition:

```
expr(R) ⟶ term(T) , [+], expr(E), {R is T+E}.
expr(R) ⟶ term(R).
```

In **Functional Programming**, input values must be parameters of functions, and output values must be results of evaluating functions. Therefore, the input sentence is the only input parameter of a parser function, while the representation and the output sentence are its result value, usually in the shape of a pair `(repr,sent)`. The solutions provided to the questions of the points (i), (ii) and (iii) mentioned above are:

(i) Since non-determinism is not a built-in mechanism of functional languages, the alternative of parsers need to be 'simulated'. This problem can be solved by using the so called *list of successes* (see [Wad85]). The idea is to collect in a list the results of trying each alternative, hence representing different alternatives through different elements of the list. In such a context, an empty list means that the parser has failed.
Therefore the type of FP parsers is `type Parser rep sym = [sym] → [ (rep,[sym]) ]`. The alternative operator can be defined now as: `(p <|> q) s = (p s) ++ (q s)`.
(ii) The sequence of parsers can be defined by a suitable HO combinator <∗> , which hides the passing of the output sentence of each parser as input sentence of the next one. However, the definition of <∗> depends also on the representation, as explained in the following point.

(iii) Often the representation of a parser function must be defined in relation to the representations of its components. In FP this is achieved through the definition of the sequence operator: `(p <∗> q) s = [ (y,s'') | (x,s') ← p s, (y,s'') ← (q x) s' ]` which has to deal with representations and not only with output sentences. In order to build the parser representation, the operator <∗> takes out the representation of the parser `p`, which is used as input argument for the function `q`, usually a lambda abstraction.

These operators, together with suitable declarations of priority and associativity, allow us to define

```
expr = term <∗> λ t → plus_minus <∗> λ o → expr <∗> λ e → return (o t e)
     <|>  term
```

where `t` stands for a number representing the evaluation of `term`, `o` stands for a functional value (+ or −) and `e` denotes the result of evaluating the expression of the right-hand side. Function `return` can be defined as `return x s = [(x,s)]`. If we consider parsers as *monads* (see [Wad95]), this notation can be abbreviated by using the *do notation* [Lau93,HM97], which is provided as a syntactic sugar to combine monads in sequence:

```
expr = do { t ← term, o ← plus_minus, e ← expr, return (o t e) }
     <|>  term
```

3.2 Parsers in $\mathcal{TOY}$

Now we are ready to define our model of parsers. In $\mathcal{TOY}$, we distinguish between parsers without representation, which we will call simply *parsers,* and *parsers with representation.* Parsers in $\mathcal{TOY}$ have the simple type:

```
parser Sym = [Sym] → [Sym]
```

that is, they take a sentence and return the non-consumed part of the sentence. Usually `Sym` stands for `char`, but in section 5.3 we will see an example of a language whose sentences are lists of integer numbers.

(i) The solution for the alternative of parsers provided by LP is simpler than that of FP. However, the introduction of the HO combinator <|> in FP permits a more expressive notation. In $\mathcal{TOY}$ we can combine the expressiveness of HO combinators of FP with the simpler definitions allowed by non-deterministic features of logic languages: A suitable non-deterministic HO combinator <|> can be defined in $\mathcal{TOY}$ as

```
(P <|> Q) Sent = P Sent
(P <|> Q) Sent = Q Sent
```

(ii) The definition of a sequence combinator <∗> in FP avoids the introduction of *ad hoc* mechanisms such as DCG's. However the definition of <∗> is complicated as it must take care of representations. In $\mathcal{TOY}$ we also define a combinator <∗> for sequence, but using the straightforward definition:

```
(P1 <*> P2) I = P2 O1 ⟸ P1 I == O1
```

that is, the first parser `P1` is applied to the input sentence `I`, and then the second parser is applied to the value `O1` returned by `P1`.

(iii) The solution provided by LP for handling representations is much simpler than the FP solution. Therefore, the representations in $\mathcal{TOY}$ will be extra (usually output) arguments of the parser functions. The type of parsers with representation is:

```
type parser_rep Rep Sym = Rep → parser Sym
```

It is worth noticing that if `P` is of type `parser_rep` then `P R` will be of type `parser`. Hence, we do not need to define an special sequence combinator for parsers with representation: the operator <∗> can also be used in such situations.

An alternative combinator for parsers with representation is, however, necessary. It can be easily defined in terms of <|> as: `(P1 <‖> P2) Rep = P1 Rep <|> P2 Rep`, meaning that the alternative of values of type `parser_rep` is converted into an alternative of values of type `parser` as soon as the representation `Rep` is provided.

As a convenient tool for attaching representations to parsers we define the combinator >>, which converts a `parser` in a `parser_rep`, as:

```
( >> )::parser A → B → parser_rep B A
(P >> Expr) R I = O ⟸ P I == O, Expr == R
```

That is, the variable `R` standing for the representation is matched with the expression `Expr` after applying the parser to the input sentence.

Before ending this section we declare the precedence of the combinators <∗>, >>, <|> and <‖>, together with their associativity. These definitions allow one to omit unnecessary parentheses.

```
infixr 40 <*>        infixr 30 >>        infixr 20 <|>, <‖>
```

4 Simple parsers and combinators

In this section we introduce a set of simple parsers and parser combinators that we will use to build more complicated parsers later. They are also our first examples of parsers in $\mathcal{TOY}$ and are based on the FP parsers described in [Fok95,Hut92].

The simplest parser, `empty`, recognizes the empty sentence, which is a prefix of every sentence. Hence, `empty` always succeeds without consuming any prefix of its input:

```
empty:: parser A
empty S = S
```

Parser `terminal T` recognizes a single symbol `T`, failing otherwise.

```
terminal:: A → parser A
terminal T [T|L] = L
```

Sometimes it is desirable to recognize not a fixed symbol, but any one fulfilling a given property `P`. Function `satisfy` accomplishes this aim:

```
satisfy:: (A → bool) → parser_rep A A
satisfy P X [X|L] = if P X then L
```

Notice that `satisfy P` is a parser with representation, as it returns as representation the recognized terminal `X`. For instance, the next parser recognizes either a letter or a digit:

```
alpha = satisfy is_letter  <‖>  satisfy is_digit
```

assuming suitable definitions for `is_letter` and `is_digit`.

In section 3 we introduced some parser combinators: <*>, <|> and >>. Here we introduce two new ones: `star` and `some`. Combinator `star` represents the repetition zero or more times of the parser with representation `P`. The representation retrieved by `star P` is a list collecting the representations of each repetition of `P`.

```
star:: parser_rep A B → parser_rep [A] B
star P  =  P X <*> (star P) Xs  >> [X|Xs]
      <‖> empty                 >> []
```

Function `some` represents the repetition at least once of the same parser, and can be defined easily in terms of `star`: `some P = P X <*> star P Xs  >> [X|Xs]`.

5 Examples

This section is devoted to present some examples of parsers in $\mathcal{TOY}$. We intend to show how, by means of the simple basic parsers and parser combinators defined before, we achieve the same expressiveness as FP parsers. Furthermore, interesting capabilities of LP parsers, such as the possibility of generating sentences instead of recognizing them, are preserved.

```
expression,term,factor,num,digit::parser_rep real char
digit::parser_rep char char
plus_minus,prod_div::parser_rep (real→ real→ real) char

expression  =  term T <*> plus_minus Op <*> expression E  >> (Op T E)
           <||> term

term        =  factor F <*> prod_div Op <*> term T  >> (Op F T)
           <||> factor

factor      =  terminal '(' <*> expression E <*> terminal ')'  >> E
           <||> num

plus_minus  =  terminal '+'  >> (+)
           <||> terminal '-'  >> (-)
prod_div    =  terminal '*'  >> (*)
           <||> terminal '/'  >> (/)

num         =  some digit L  >> (numeric_value L)
digit       =  satisfy is_digit
```

Fig. 1. Parser for arithmetic expressions

5.1 Arithmetic Expressions

The parser showed in figure 1 recognizes arithmetic expressions made from integer numbers, the operators +, -, *, / and parentheses. The main parser is `expression` which returns as representation the numeric value of the expression. The first rule says that an expression is either a term followed by an operator + or - and ended by another expression or simply a term. In the first case the combinator >> shows that the representation of the expression is the result of applying the representation of the operator to those of the two other components. In the second case the representation of the expression is the representation of the term. Among the rest of the parsers, we must point out the introduction of a function `numeric_value` which converts a string of digits into its numeric value. The definition of this function relies on the standard functions `foldl1` and `map`: `numeric_value L = foldl1 ((+).(10*)) (map val L)` and constitutes a typical example of how FLP inherits the higher-order machinery usual in FP. For instance, the goal `expression R "(10+5*2)/4" == []` succeeds with `R == 5`.

5.2 Parsers as generators

Owing to the possibility of including logical variables in goals, FLP parsers may be regarded as generators as well as recognizers. Consider for instance the parser

```
palin = empty  <|>  a  <|>  b  <|>  a <*> palin <*> a  <|>  b <*> palin <*> b
a     = terminal 'a'
b     = terminal 'b'
```

which recognizes the language of the palindrome words over the alphabet $\Sigma = \{a, b\}$. Using this parser we may 'ask' for sentences of length two in the language recognized by `palin`:

```
palin [X,Y] == []
```

Two answers are retrieved, namely `X='a', Y='a'` and `X='b', Y='b'`, meaning that *"aa"* and *"bb"* are the only words of length two in this language.

5.3 Numerical Constraints

The growing interest in languages representing spatial relationships (e.g. visual languages [HMO91]) has introduced the study of *numerical constraints* in relation to the parsing problem. Here we show a very simple but suggestive example of how our parsers can integrate numerical constraints easily.

Suppose we are interested in a parser for recognizing *boxes*, regarding a box as a rectangle whose sides are parallel to the X and Y axes. The terminals of the language will be pairs of integers representing points in the plane, and a valid sentence will be a sequence of four points standing for the corners of the box, beginning with the lower-left and following anti-clockwise. The desired representation is a pair of points representing the lower-left and the upper-right corners of the box.

```
box:: parser_rep ((real,real),(real,real)) (real,real)
box   = point (X1,Y1) <*> point (X2,Y2) <*>
        point (X3,Y3) <*> point (X4,Y4)  >> ((X1,Y1),(X3,Y3))
        <== Y1==Y2, X1==X4, X2==X3,Y4==Y3, Y1<Y4, X1<X2

point:: parser_rep (real,real) (real,real)
point = terminal (X,Y)  >> (X,Y)
```

The conditions assure that the points actually represent a box. Note that these constraints are settled before parsing the point. As a consequence, if the points do not have the shape of a box, the parser can fail as soon as possible. For instance, if the condition `Y1==Y2` is not verified, the parser will fail just after parsing the second point.

For our example to work properly, the language must be able to handle numerical constraints concerning still uninstantiated variables, and to check incrementally the accumulated constraints whenever new ones are imposed during the computation. Such an extension of the language considered so far is described in [AH+96], and is actually implemented in the system $\mathcal{TOY}$ (with such this example is indeed executable). For example we can fix two points of the box and ask $\mathcal{TOY}$ for the conditions that the other two points must satisfy to form a box:

```
box R [(1,2), (4,2), P, Q] == []
```

The goal succeeds, and $\mathcal{TOY}$ returns the following answer:

```
R == ((1, 2), (4, _A))   P == (4, _A)   Q == (1, _A)   {_A>2.0 }
```

which are the equations that the variables must satisfy, including an arithmetical constraint.

6 Parsers as data

In previous sections the advantages of defining parsers in $\mathcal{TOY}$ have been discussed. Here we intend to show how functional-logic languages allowing higher-order patterns can consider parsers as *truly first class data values*, in a broader sense of the term than usual. It is worthwhile to point out that the following discussion is held in a purely declarative framework.

6.1 The structure of $\mathcal{TOY}$ parsers

Consider the parser `ab` defined as: `ab = terminal 'a' <*> terminal 'b'` Function `ab` can be reduced directly to its right-hand side (`terminal 'a' <*> terminal 'b'`), while `<*>` and `terminal` need to be applied to the input sentence in order to be reduced. Therefore we can say that the 'structure' of `ab` is of the shape `A <*> B`, where both `A` and `B` are of the form `terminal T`. The interesting point is that `A <*> B` and `terminal T` are valid HO patterns (see Sec. 2). In general, any parser `P` defined through the *basic components* {`<*>` , `<|>`, `>>` , `empty`, `satisfy`, `terminal`}, can be decomposed by matching it against suitable HO patterns. In this context, the basic components can be effectively considered as *data constructors*, and parsers as *data values*.

6.2 Folding parsers

Functions *fold* are widely used in FP. They replace the constructors of structures by given functions. The most usual examples of these constructions are those that handle lists, i.e. the standard functions `foldr`, `foldl`, ..., but the same technique can be applied to structures other than lists (see [MJ95] for many examples). As we have shown above, parsers can be considered data values, and hence we can define a function `fold_p` that replaces the constructors of a parser by arbitrary functions. The definition is a little bit tedious, but straightforward:

```
fold_p (Empty,_,_,_,_,_) empty         = Empty
fold_p (_,Term,_,_,_,_)  (terminal T)  = Term T
fold_p (_,_,Sat,_,_,_)   (satisfy P R) = Sat P R
fold_p F@(_,_,_,Seq,_,_) ((<*>) A B)   = Seq (fold_p F A ) (fold_p F B)
fold_p F@(_,_,_,_,Alt,_) ((<|>) A B)   = Alt (fold_p F A) (fold_p F B)
fold_p F@(_,_,_,_,_,Rep) (( >> ) A B R) = Rep (fold_p F A) B R
```

The first argument of `fold_p` is a tuple of the form `(Empty, Term, Sat, Seq, Alt, Rep)` with the functions that will replace the 'constructors' `empty`, `terminal`, `satisfy`, `<*>` , `<|>` and `>>` respectively. The second argument is the parser we want 'fold'. Function `fold_p` is applied recursively to the structure of the parser, replacing each constructor by its correspondent function. At first sight, function `fold_p` might seem rather unuseful. Indeed, most of the parsers definitions are recursive, and hence their basic structures are infinite: function `fold_p` would peer into such structures forever, if normal forms were being looked for. Instead, we will show that, due to lazy evaluation, function `fold_p` allows us to check interesting properties of the represented grammar, such as ambiguity. Observe that (apart from syntax details) the definition of `fold_p` is by no means a valid *Haskell*program, due to the presence of HO patterns.

6.3 Checking ambiguity

We say that a grammar specification is ambiguous when a sentence exists with more than one parse tree. Consider for instance the grammar represented by the following $\mathcal{TOY}$ parser (we have labelled the productions with P1, P2 and P3):

```
s = terminal 'i' <*> s <*> terminal 'e' <*> s (P1)
    <|> terminal 'i' <*> s                     (P2)
    <|> terminal 'o'                           (P3)
```

This grammar is ambiguous, since the sentence *iioeo* can be derived following either the left derivation *P1, P2, P3, P3* or *P2, P1, P3, P3*. As ambiguity is not a nice property when defining grammars for programming languages, we would like to define a function that look for ambiguous words. A possible solution is to define a parser with representation `s'`, whose representation is the parse tree. By using `s'` we can look for sentences with two different representations.

However this means that we need to define a parser `P'` each time we want to study the ambiguity of a parser *P*. Obviously, it is much better to mechanize this process by defining a suitable function `build_tree` that converts a given parser into a new parser that returns as representation the parser tree. This function can be defined in terms of `fold_p` as follows:

```
build_tree :: parser A → parser_rep [int] A
build_tree = fold_p (empty_t,term_t,sat_t,seq_t,alt_t,rep_t)
```

where

```
empty_t          =  empty                >> []
term_t T         =  terminal T           >> []
sat_t P RI       =  satisfy P RI         >> []
seq_t P1 P2      =  P1 R1 <*> P2 R2      >> R1++R2
alt_t P1 P2      =  P1 R                 >> [1|R]
                <|> P2 R                 >> [2|R]
rep_t P Expr Rep =  (P Tree  >> Expr) Rep >> Tree
```

To represent the parse tree we use a list which collects the chosen alternatives. In the case of `empty`, `terminal` and `satisfy` all we return as representation is the empty list. In order to understand the sequence and the alternative we must keep in mind that `P1` and `P2` have been 'folded' already, i.e they are parsers with representation. The sequence applies each parser and concatenates the two resulting lists, while the alternative includes the number of the chosen option in the current representation. Finally, `rep_t` takes charge of the parsers with representation. It first applies the parser, getting the list of alternatives. Then the parser is again converted into a parser with representation, in order to keep the initial representation unaffected. For instance, the goal `build_tree s L "iioeo" == []` succeeds with `R == [ 1, 2, 1, 2, 2, 2, 2 ]` as well as with `R == [ 2, 1, 1, 2, 2, 2, 2 ]`. Owing to the right associativity of <|>, the sequence `2,1` stands for `(P2)`, while `2,2` means that the the production `(P3)` was applied.

To get the list of alternatives of a `parser_rep` it must be converted into a `parser`, as in the goal `build_tree (expression R) L "1+2" == []` . The application the `parser_rep expression` to `R` yields a value of type `parser`, reduces the occurrences of <‖> into <|>, and provides the extra argument that the HO pattern `(( >> ) A B R)` requires in the definition of `fold_p`. This goal succeeds with `R==3, L==[ 1, 2, 2, 2, 1, 2, 2, 2, 2 ]` as its only solution, meaning that the sentence has exactly one parse tree.

Now it is easy to define a function that looks for words with two different parser trees:

```
ambi:: parser A → [A]
ambi P = W ⇐ gen_word==W, build_tree P R1 W==[], build_tree P R2 W==[],
             not (R1==R2)
```

The first condition is used to generate general words in a non-deterministic fashion (see below). The next two conditions try to parse the word twice, while the third condition assures that the two representations returned are different. If they are equal, then backtracking is enforced and a new parse is tried. Otherwise the word `W` admits two different parse trees (i.e. the grammar is ambiguous) and `W` is returned. Observe that we are using here the LP capabilities of our language, since `R1` and `R2` are new (existential) variables. It is also worth noticing that both the fold and the parsing of the input sentence are performed at the same time, avoiding the folding of unsuccessful (and infinite) branches of the parser tree.

Since ambiguity is a semidecidable property, we should only check words in a given range of length. If we find that the parser is ambiguous for any of these words, we have demonstrated that it is ambiguous. Otherwise, we can either try a wider range of lengths, or accept this partial result of non-ambiguity. Owing to this we define a non-deterministic function `word_from_to` which generates all the general sentences, i.e. lists of variables, whose length is less than or equal to a given number `N`.

```
all_words N = []
all_words N = [_|all_words (N-1)] ⇐ N >= 0
```

Thus we can define `gen_words`, for instance, as: `gen_words = all_words 10`. At this point we can check that `s` is ambiguous by trying the goal `ambi s == W` which succeeds with `W == "iioeo"`. Conversely, similar goals for the parser with representation `expression` or for the parser `palin`, both defined in Sect. 5, will fail, meaning that there is no expression or palindrome word whose length is less than or equal to 10 with two different parse trees.

7 Conclusions

This paper shows how a functional-logic language supporting non-deterministic functions allows defining parsers which combine most of the nicest properties of both functional and logic parsers. Our approach has been presented by means of a concrete language, $\mathcal{TOY}$, but other functional-logic languages supporting non-deterministic functions, like *Curry* [Han99], could have been used. Specifically, the expressiveness of $\mathcal{TOY}$ parsers is akin to that of FP parsers, but based on simpler concepts and definitions. This is due to the adoption in our model of typical LP characteristics, like the natural way of handling non-determinism provided by non-deterministic computations. Also, parsing in $\mathcal{TOY}$ benefits from the use of logical variables to return representations, thus avoiding the introduction of monads and lambda abstractions. Actually, this technique can be generalized, and represents a simple and natural FLP alternative to monads in many situations (see [CL99]). Despite their 'functional' shape, parsers in $\mathcal{TOY}$ share with parsers of LP the possibility of multiple modes of use, generating as well as recognizing sentences. We have also investigated further possibilities of our approach to parsers, making use in this case of more specific features of TOY. First, we have briefly indicated how these parsers benefit from the inclusion of arithmetical constraints. In a different direction, $\mathcal{TOY}$'s possibility of using HO patterns in heads of rules has given parsers the role of data values in a very strong sense. We have defined a *fold* function for them, and used this function to check the ambiguity of grammars. Other interesting properties of the underlying grammars, such as the $LL(1)$ property, can be also checked using the same technique, as shown in [CL98a].

Acknowledgements:
We thank Mario Rodríguez-Artalejo for many valuable comments about this work.

References

[AH+96] P. Arenas-Sánchez , T. Hortalá-González, F.J. López-Fraguas, E. Ullán-Hernández. *Functional Logic programming with Real Numbers*, in M. Chakavrarty, Y. Guo, T. Ida (eds.) *Multiparadigm Logic Programming*, Post-Conference Workshop of the JICLP'96, TU Berlin Report 96-28, 47–58, 1996.

[AR97] P. Arenas-Sánchez, M. Rodríguez-Artalejo. *A Semantic Framework for Functional Logic Programming with Algebraic Polymorphic Types.* Procs. of CAAP'97, Springer LNCS 1214, 453–464, 1997.

[CLS97] R. Caballero-Roldán, F.J. López Fraguas, J. Sánchez-Hernández. *User's Manual For $\mathcal{TOY}$* . Technical Report D.I.A. 57/97, Univ. Complutense de Madrid 1997. The system is available at `http://mozart.sip.ucm.es/incoming/toy.html`

[CL98a] R. Caballero-Roldán, F.J. López-Fraguas. *Functional-Logic Parsers in $\mathcal{TOY}$* . Technical Report S.I.P. 74/98. Univ. Complutense de Madrid 1998.
Available at `http://mozart.sip.ucm.es/papers/1998/trparser.ps.gz`

[CL99] R. Caballero, F.J. López-Fraguas. *Extensions: A Technique for Structuring Functional-Logic Programs.* Procs. of Perspectives of System Informatics, 1999, Springer LNCS. To appear.

[Fok95] J. Fokker. *Functional Parsers.* In J. Jeuring and E. Meijer editors, Lecture Notes on Advanced Functional Programming Techniques, Springer LNCS 925, 1995.

[GH+99] J.C. González-Moreno, T. Hortalá-González, F.J. López-Fraguas, M. Rodríguez-Artalejo. *An Approach to Declarative Programming Based on a Rewriting Logic.* Journal of Logic Programming, Vol 40(1), july 1999, pp 47–87. A preliminary version appeared under the title *A Rewriting Logic for Declarative Programming* in Procs. of ESOP'96, Springer LNCS 1058.

[GHR93] J.C. González-Moreno, T. Hortalá-González, M. Rodríguez-Artalejo. *On the Completeness of Narrowing as the Operational Semantics of Functional Logic Programming.* Procs. of CSL'92, Springer LNCS 702, 216–230, 1993.

[GHR97] J.C. González-Moreno, T. Hortalá-González, M. Rodríguez-Artalejo. *A Higher Order Rewriting Logic for Functional Logic Programming.* Procs. of ICLP'97, The MIT Press, 153–167, 1997.

[Gon93] J.C. González-Moreno. *A Correctness Proof for Warren's HO into FO Translation.* Procs. of GULP'93, 569–585, 1993.

[Han94] M. Hanus. *The Integration of Functions into Logic Programming: A Survey.* J. of Logic Programming 19-20. Special issue "*Ten Years of Logic Programming*", 583–628, 1994.

[Han99] M. Hanus (ed.). *Curry, an Integrated Functional Logic Language*, January 1999. Available at `http://www-ir.informatik.rwth-aachen.de/ hanus/curry/report.htlm`

[HAS98] *Report on the Programming Language Haskell 98: a Non-strict, Purely Functional Language.* Simon Peyton Jones and John Hughes (eds.), February 1999.

[Hut92] G. Hutton. *Higher-Order Functions for Parsing.* J. of Functional Programming 2(3):323-343, July 1992.

[HMO91] R. Helm, K. Marriot and M. Odersky. *Building Visual Languages Parsers.* ACM CHI'91. ACM Press 1991, pp. 105-112

[HM97] G. Hutton, E. Meijer. *Functional Pearls. Monadic Parsing in Haskell.* Journal of Functional Programming 8 (4), 1998, 437-444.

[Lau93] J. Launchbury. *Lazy imperative programming.* In Procs. ACM Sigplan Workshop on State in Programming Languages, YALE/DCS/RR-968, Yale University, 1993.

[LLR93] R. Loogen, F.J. López-Fraguas, M. Rodríguez-Artalejo. *A Demand Driven Computation Strategy for Lazy Narrowing.* Procs. of PLILP'93, Springer LNCS 714, 184–200, 1993.

[LS99] F.J. López-Fraguas, J. Sánchez-Hérnandez. *$\mathcal{TOY}$: A Multiparadigm Declarative System.* To appear in Proc. RTA'99, Springer LNCS.

[MJ95] E. Meijer, J. Jeuring. *Merging Monads and Folds for Functional Programming.* Adv. Functional Programming Int. School, Baastad (Sweden). Springer LNCS 925, 228–266, 1995.

[Pre96] C. Prehofer. *Solving Higher-Order Equations. From Logic to Programming. Progress in Theoretical Computer Science*, Birhäuser, 1998.

[SS86] L. Sterling, E. Shapiro. *The Art of Prolog*, The MIT Press, 1986.

[Wad85] P. Wadler. *How to Replace Failure by a List of Successes*, Proc. IFIP FPCA'85, Springer LNCS 201, 1985, 113–128.

[Wad95] P. Wadler. *Monads for functional programming.* In J. Jeuring and E. Meijer editors, Lecture Notes on Advanced Functional Programming Techniques, Springer LNCS 925. 1995