Dependency grammars as Haskell programs

Tomasz Obrębski
Adam Mickiewicz University
Poznań, Poland
obrebski@amu.edu.pl

Abstract

In the paper we try to show that a lazy functional language such as Haskell is a convenient framework not only for implementing dependency parsers but also for expressing dependency grammars directly in the programming language in a compact, readable and mathematically clean way. The parser core, supplying necessary types and functions, is presented together with two examples of grammars: one trivial and one more elaborate, allowing to express a range of complex grammatical constraints such as long distance agreement. The complete Haskell code of the parser core as well the grammar examples is included.

1 Introduction

Functional programming is nowadays probably the most rapidly developing subfield in the domain of theory and implementation of programming languages. Functional languages, with Haskell as their flagship, are continuously evolving, mostly by absorbing more and more mathematics (abstract algebra, category theory). This translates into their increasing expressiveness, which is directly usable by programmers.

The combination of keywords *functional programming* and *parsing* usually brings to mind the monadic parsing technique (Hutton and Meijer, 1998) developed as an attractive functional offer for parser builders. This technology is dedicated mostly to artificial languages. Much less work has been done in functional programming paradigm regarding natural language parsing technologies. The outstanding exception is the *Grammatical Framework* environment (Ranta, 2011). Written in Haskell with extensive use of higher-order abstraction and laziness property, it offers impressive capabilities of making generalizations in all conceivable dimensions in a large and highly multilingual language model including morphological, syntactic and semantic layers. Some other works, which may be mentioned here, are due to Ljunglöf (2004), de Kok and Brouwer (2009), Eijck (2005).

As far as dependency-based parsing and language description is concerned (Kubler et al., 2009), the author is not aware of any attempts to apply functional programming techniques.

Below we try to show that a lazy functional language such as Haskell is a convenient framework not only for implementing dependency parsers but also for expressing dependency grammars directly as Haskell code in a compact, readable, and mathematically clean way.

A question may arise: why the ability to write a grammar directly in a programming language should be considered desirable. There are already plenty of grammatical formalisms to choose from. And what makes Haskell more interesting target language then others, e.g. Python. The answer to the first questions is: (1) the grammar writer is free in choosing the way the lexical and grammatical description is formulated; (2) full strength of the language may be used according to the needs. DCG grammars (Pereira and Warren, 1980) are a good example here. The answer to the second one is: (1) the grammar may be expressed in declarative way in the language very close to that of mathematics, in terms of basic mathematical notions such as sets, functions, and relations; (2) functional character of Haskell allows for making generalizations wherever the grammar writer finds it advantageous; (3) Haskell syntax allows for formulating grammatical statements in a compact, clean, mathematical manner; (4) Haskell libraries supply support for mathematical objects frequently used in language description, e.g. lattices (cf. Koster, 1992; Levy and Pollard, 2001),

D S Sharma, R Sangal and A K Singh. Proc. of the 13th Intl. Conference on Natural Language Processing, pages 71–80,
Varanasi, India. December 2016.

semirings (cf. Goodman, 1999), to mention just two.

2 The Haskell toolbox

Haskell is a purely functional programming language, applying lazy evaluation strategy, see (Jones, 2002) for language specification, (Lipovaca, 2011) for introductory course, and (Yorgey, 2009) for information on advanced Haskell algebraic type classes. We will take a closer look at two Haskell types on which the parser and grammar implementation is based:

`[a]` – list of elements of type `a`

`a →[a]` – a function taking an argument of type `a` and returning a list of elements of type `a`.

Below, we are going inspect the properties of those types as well as functional tools which will allow us to operate on them conveniently.

Lists are used to store collections of values. Their interpretation depends on the context. We use lists for representing sequences, sets, alternatives of values as well as possible lack of a value (singleton list - value exists, empty list - no value). The other important type is the type of functions that take an argument of some type `a` and return a list of values of type `a`, i.e. the type `a →[a]`. These are functions which return sequences, sets, alternatives, or a possibly lacking value, all represented by lists. Here are some examples:

- the function which extends the parse with a new node produces several alternative results for the word "fly" (type `Parse →[Parse]`);
- the function returning the preceding node has no value for the first node (type `Node →[Node]`);
- the function computing transitive heads of a node returns a set of nodes (type `Node →[Node]`).

A list type `[a]` is obtained by applying the list functor `[]` to some type `a`, with no constraints on what `a` is. Two properties are particularly useful: (1) the list functor `[]` is an instance of `Monad`; (2) a list of elements of any type is an instance of `Monoid`.

Functions and operators from both classes (`Monoid` and `Monad`) may be intermixed for lists because they share the value-combining operation: both the *join*[1] operation in the list monad and the operation in the monoid `[a]` is concatenation. An important consequence of the fact that `[a]` is an instance of `Monoid` is that all functions which return `[a]` are also instances of `Monoid`. Below we summarize the list of operators on values of type `[a]` and `a -> [a]`, supplied by classes `Monad` and `Monoid`, which we will make use of.

- $\oplus$ :: `[a]` → `[a]` → `[a]`
  (`instance Monoid [a]`)

  `xs` $\oplus$ `ys` combines values contained in `xs` with those in `ys` producing one list with both `xs` and `ys`;

- $\oplus$ :: `(a →[a])` → `(a →[a])` → `(a →[a])`
  (`instance Monoid (a -> [a])`)

  `f` $\oplus$ `g` combines functions `f` and `g`, which both return a list of values of type `a`, into a single function returning one list that contains the return values of both `f` and `g`;

- `>=>` :: `(a →[a])` → `(a →[a])` → `(a →[a])`
  (`instance Monad []`)

  `f >=> g` composes functions `f` and `g` in one function (of the same type). The resulting function applies `g` to each value from the list returned by `f` and returns the results combined in one list;

- `>>=` :: `[a]` → `(a →[a])` → `[a]`
  (`instance Monad []`)

  `[a] >>= f` applies `f` to all values from `[a]` and combines the results.

In addition to the operators listed above, we will use the function:

- `pure` :: `a →[a]`
  (`instance Applicative []`)

  which returns a singleton list containing its argument. It is equivalent to the monadic `return` function, but it reads much better in our contexts.

Having a function of type `a →[a]`, we will be frequently also interested in its transitive closure or reflexive transitive closure (strictly speaking - the term *transitive closure* refers to the underlying

[1] The operation which transforms a list of lists into a flat list.

relation). We will introduce the family of closure functions. They are used for example to obtain the function which computes transitive heads from the function that returns the head.

```
clo,mclo,rclo,mrclo :: (a → [a]) → a → [a]
clo f     = f >=> ( pure ⊕ clo f )
rclo f    = pure ⊕ clo f
mclo f    = f >=> mrclo f
mrclo f x = let fx = f x in if null fx
                            then pure x
                            else fx >>= mrclo f
```

The function `clo` computes the closure of its argument function `f`. The function `f` is (Kleisli) composed with the function which combines ($\oplus$) its arguments (`pure` values of `f`) with the values of recursive application of `clo f` on each of those arguments. The function `rclo` computes the reflexive transitive closure of its argument function `f`. The argument itself (`pure`) is combined ($\oplus$) with the values returned by `clo f` applied to this argument. The `m*` versions of `clo` and `rclo` return only maximal elements of closures, i.e. those for which the argument function `f` returns no value.

The operators and functions presented above will be expecially useful for working with relations. This is undoubtedly the most frequently used mathematical notion when talking about dependency structures. We use relations to express relative position of a word (node) with respect to another word (predecessor, neighbor, dependent, head). We also frequently make use of such operations on relations as transitive closure transitive head, reflexive transitive dependent) or composition (transitive head of left neighbour).

Haskell is a functional language. We will thus have to capture operations on relations by means of functions. The nearest functional relatives of a relation are image functions.

Given a relation $\mathbf{R} \subset A \times B$, the image functions $\mathbf{R}[\cdot]$ are defined as follows (1 – image of an element, 2 – image of a set):

$$\begin{array}{lllll} (1) & \mathbf{R}[x] & = & \{y \mid xRy\} & \text{where } x \in A \\ (2) & \mathbf{R}[X] & = & \{y \mid x \in X \wedge xRy\} & \text{where } X \subset A \end{array}$$

Haskell expressions corresponding to image functions and their use are summarized below (`x` has type `a`, `xs has type [a]`, `r` and `s` have type `a →[a]`):

$\mathbf{R}[x]$	`r x`	(or `pure x >>= r`)
$\mathbf{R}[X]$	`xs >>= r`	
$(\mathbf{R} \circ \mathbf{S})[\cdot]$	`s >=> r`	
$(\mathbf{R} \cup \mathbf{S})[\cdot]$	`r ⊕ s`	
$\mathbf{R}^+[\cdot]$	`clo r`	
$\mathbf{R}^*[\cdot]$	`rclo r`	

3 Data structures

The overall design of the parser is traditional. Words are read from left to right and a set of alternative parses is built incrementally. We start with describing data types on which the parser core is based. They are designed to fit well into the functional environment[2].

3.1 The `Parse` type

A (partial) parse is represented as a sequence of parse steps. Each step consumes one word and introduces a new node to the parse. It also adds all the arcs between the new node and the nodes already present in the parse. All the data added in a parse step – the index of the new node, its category and the information on its connections with the former nodes – will be encapsulated in a value of type `Step`:

```
type Parse = [Step]
data Step  = Step Ind Cat [Arc] [Arc]
             deriving (Eq,Ord)
```

A value of type `Step` is built of the type constructor of the same name and four arguments:

(1) the word's index of type `Ind`. It reflects the position of the word in the sentence. It is also used as the node identifier within a parse;

(2) the syntactic category of the node represented by a value of type `Cat`;

(3) the arc linking the node to its left head. This value will be present only if the node is preceded by its head in the surface ordering. List is used to represent a possibly missing value.

(4) the list of arcs which connect the node with its left dependents.

We also make `Step` an instance of the classes `Eq` and `Ord`. This will allow us to use comparison operators (based on node index order) with values of type `Node` introduced below.

The value of type `Arc` is a pair:

[2]The functional programming friendly representation of a parse was inspired by (Erwig, 2001).

```
type Arc = (Role,Ind)
```

where `Ind` is the integer type.

```
type Ind = Int
```

For the sentence *John saw Mary.* we obtain the following sequence of the parse steps:

```
[  Step 3 N [(Cmpl,2)] [],
   Step 2 V [] [(Subj,1)],
   Step 1 N [] []          ]
```

We introduce three operators for constructing a parse:

```
infixl 4 <<, +->, +<-
(<<) :: Parse -> (Ind,Cat) -> Parse
p << (i,c) = Step i c [] [] : p

(+->),(+<-) :: Parse -> (Role,Ind) -> Parse
(Step i c [] d: p) +-> (r,j) = Step i c [(r,j)] d  : p
(Step i c h  d: p) +<- (r,j) = Step i c h ((r,j):d) : p
```

The operator `<<` adds an unconnected node with index `i` and category `c` to the parse `p`. The operator `+->` links the node `i` as the head of the current (last) node with depencency of type `r`. The operator `+<-` links the node `i` as the dependent of the current node with depencency of type `r`. All the operators are left associative and can therefore be sequenced without parentheses. The expression constructing the parse for the sentence *John saw Mary.* is presented in Figure 1.

3.2 The `Node` type

In i-th step, the parser adds the node i to the parse and tries to establish its connections with nodes $i-1$, $i-2$, ..., 1. In order to make a decision whether the dependency of type r between nodes i and j, $j < i$, is allowed, various properties of the node j has to be examined. They depend on the characteristics of the grammar. Some of them are easily accessible, such as the node's category. Other ones are not accessible directly, such as e.g. the set of roles on outgoing arcs, categories of dependent nodes.

When the parser has already performed n steps, full information on each node $i, i < n$, including its connections with all nodes $j, j < n$, is available. In order to make this information accessible for the node i, we use the structure of the following type for representing a node:

```
data Node  = Node [Step] [Step] deriving (Eq,Ord)
```

The first list of steps is the parse history up to the step i. The second list contains the steps which follow i, arranged from the one directly succeeding i up to the last one in the partial parse.

The node representation contains the whole parse, as seen from the node's perspective. The redundancy in the node representation, resulting from the fact that the whole parse is stored in all nodes during a computation, is apparent only. Lazy evaluation guarantees that those parts of the structure, which will not be used during the computation, will never be built. Thus, we can see a value of type `Node`, as representing a node equipped with the potential capability to inspect its context. In the last node of a parse, the *history* list will contain the whole parse and the *future* list will be empty.

```
lastNode :: Parse → Node
lastNode p = Node p []
```

The following functions will simplify extracting information from a `Node` value.

```
ind :: Node → Ind
ind (Node (Step i _ _ _ : _) _) = i
cat :: Node → Cat
cat (Node (Step _ c _ _ : _) _) = c

hArc, dArcs :: Node → [Arc]
hArc (Node (Step _ _ h _ : _) _) = h
dArcs (Node (Step _ _ _ d : _) _) = d
```

The most essential property of a `Node` value is probably that all the other nodes from the partial parse it belongs to may be accessed from it.

```
lng, rng :: Node → [Node]
lng (Node (s:s':p) q)  = [Node (s':p) (s:q)]
lng _                  = []
rng (Node p (s:q))     = [Node (s:p) q]
rng _                  = []

preds, succs :: Node → [Node]
preds  = clo lng
succs  = clo rng
```

The function `lng` (left neighbour) returns the preceding node. The last step in the *history* list is moved to the beginning of the *future* list, provided that the *history* list contains at least two nodes. The function `rng` (right neighbour) does the opposite and returns the node's successor. The `clo` function was used to compute the list of predecessors and successors of a node.

The next group of functions allows for accessing the head and dependents of a node. List comprehensions allow for their compact implementation:

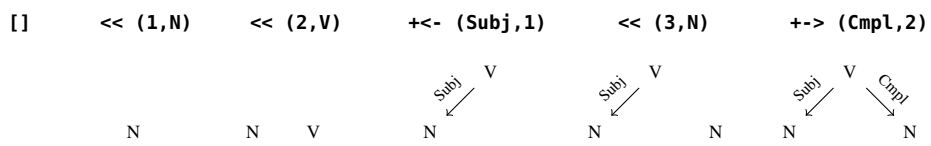

Figure 1: The expression for the parse: `[ Step 3 N [(Cmpl,2)] [], Step 2 V [] [(Subj,1)], Step 1 N [] [] ]`

```
ldp',rdp',dp',lhd',rhd',hd' :: Node → [(Role, Node)]
ldp' v = [(r,v')|v'←preds v,(r,i)←dArcs v,ind v'≡i]
rdp' v = [(r,v')|v'←succs v,(r,i)←hArc v',ind v≡i]
dp'   = ldp' ⊕ rdp'

lhd' v = [(r,v')|v'←preds v,(r,i)←hArc v,ind v'≡i]
rhd' v = [(r,v')|v'←succs v,(r,i)←dArcs v',ind v≡i]
hd'    = lhd' ⊕ rhd'
```

The function `ldp'` (left dependent) returns the list of left dependents of the node `v` together with corresponding roles: these are such elements `v'` from the list of predecessors of `v`, for which there exists an arc in `dArcs v` with index equal to the index of `v'`. To get the list of right dependents (`rdp'`) of `v`, we select those nodes from the list `succs v`, whose left head's index is equal to that of `v`. The functions `rhd'` (right head) and `lhd'` (left head) are implemented analogously. The function `dp'` which computes all dependents is defined by combining the functions `ldp'` and `rdp'` with the operator ⊕ (similarly `hd'`).

These primed functions are not intended to be used directly by grammar writers (hence their primed names)[3]. They will serve as the basis for defining the basic parser interface functions: group of functions for computing related nodes (`ldp`, `rdp`, ...), group of functions for computing roles on in- and outgoing arcs (`ldpr`, `rdpr`, ...), and finally the group of function for accessing nodes linked with dependency of a specific type (`ldpBy`, `rdpBy`, ...).

```
ldp, rdp, dp, lhd, rhd, hd :: Node → [Node]
ldp = fmap snd . ldp'
( similarly rdp, dp, lhd, rhd, hd)

ldpBy, rdpBy, dpBy :: Role → Node → [Node]
ldpBy r v = [ v' | (r,v') ← ldp' v ]
rdpBy r v = [ v' | (r,v') ← rdp' v ]
dpBy r    = ldpBy r ⊕ rdpBy r

ldpr, rdpr, dpr, lhdr, rhdr, hdr :: Node → [Role]
ldpr = fmap fst . ldp'
( similarly rdpr, dpr, lhdr, rhdr, hdr)

lhdBy,rhdBy,hdBy :: Role → Node → [Node]
lhdBy r v = [ v' | (r,v') ← lhd' v ]
rhdBy r v = [ v' | (r,v') ← rhd' v ]
hdBy r    = lhdBy r ⊕ rhdBy r
```

The functions for navigating among nodes[4] are summarized in Figure 2.

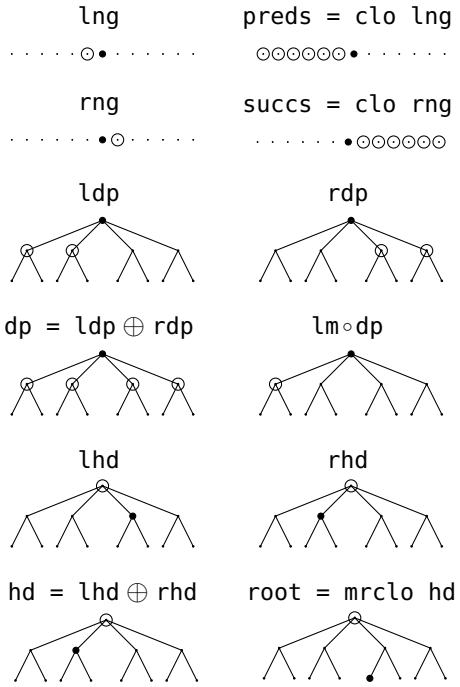

Figure 2: Node functions (black dot - the argument, circles - values)

We will end with defining three more useful functions: `lm` and `rm` for choosing the leftmost/rightmost node from a list of nodes, and `hdless` for checking whether the argument node has no head.

```
lm, rm :: [Node] -> [Node]
lm [] = []
lm xs = [minimum xs]
rm [] = []
rm xs = [maximum xs]

hdless :: Node → Bool
hdless = null ∘ hd
```

[3] They are not of type `a →[a]` and are far less usefull then e.g. functions of type `Node →[Node]` defined below (`ldp`, `rdp`, ...).

[4] Many other useful functions for navigating among parse nodes may be defined using the ones introduced above, for example: `subtree = rclo dp`, `root = mrclo hd`, `tree = root >=> subtree`, `siblings = hd >=> dp`, etc.

4 The parser core

We will begin by defining the `step` function. Given a parse and a word, it computes the next `Step`. This computation may be decomposed into two independent operations: `shift` – add a new `Step` with only word's category and the index, with no connections; and `connect` – create dependency connections for the last node in the parse. The operations `shift` and `connect` will resort to two different information sources external to the parser: the lexicon and the grammar, respectively. In the impelementation of `shift` we assume the existence of an external lexicon (see Section 5), which provides a function `dic` of type `Word -> [Cat]`. This function, given a word `w` as argument, returns a list of its syntactic categories.

```
type Word = String

shift :: Word → Parse → [Parse]
shift w p = [ p << (nextId p, c) | c ← dic w]
            where nextId [] = 1
                  nextId (Step i _ _ _ : _) = i + 1
```

The `shift` function adds to the parse `p` a new unconnected node with `w`'s syntactic category and the appropriately set index. As the word `w` may be assigned many alternative syntactic categories due to its lexical ambiguity, a list of parses is produced – one parse for each alternative reading of `w`.

In the impelementation of `connect` we assume the existence of an external grammar (see Section 5), which is required to offer the functions `heads`, `deps` of type `Node -> [(Role,Node)]` and `pass` of type `Node -> Bool`. The functions `heads` and `deps` take a node as argument and return the list of all candidate connections to heads or dependents, respectively. The `pass` function allows the grammar to perform the final verification of the complete parse (the last node is passed as the argument).

We first define two functions `addHead` and `addDep`. They add connections proposed by the grammar for the last node in the parse. The functions also check whether the candidate for the dependent node has no head attached so far.

```
addHead, addDep :: Parse → [Parse]
addHead p = [ p +-> (r,ind v') | let v = lastNode p,
                                 hdless v,
                                 (r,v') ← heads v ]
addDep p  = [ p +<- (r,ind v') | let v = lastNode p,
                                 (r,v') ← deps v,
                                 hdless v' ]
```

With these functions we can define `connect` as follows:

```
connect :: Parse → [Parse]
connect = (addDep >=> connect) ⊕ addHead ⊕ pure
```

Parses returned by `addDep`, `addHead`, are combined together with the unchanged parse (`pure`). Parses returned by `addDep` are recursively passed to `connect`, because there are may be more than one dependent to connect. The `connect` function produces parses with all possible combinations of valid connections.

Now, the step computation may be implemented by combining `shift w` and `connect`.

```
step :: Word → Parse → [Parse]
step w = shift w >=> connect
```

The whole parse will be computed (function `steps`) by applying left fold on a word list using the `step` function inside the list monad – we just have to flip the first two arguments of `step` to get the type needed by `foldM`.

```
steps :: [Word] → [Parse]
steps = foldM (flip step) []
```

Finally, the `parser` function selects complete parses (containing one tree, thus satisfying `(≡1)∘size`) and asks the grammar for final verification (`pass∘lastNode`).

```
parser :: [Word] → [Parse]
parser = filter ((≡1)∘size ∧ pass∘lastNode) ∘ steps
```

5 Lexicons and grammars

In order to turn the bare parser engine defined above into a working syntactic analysis tool we has to provide it with a lexicon and a grammar. We are short of exactly six elements: the types `Cat` and `Role`, and the functions `dic`, `heads`, `deps`, and `pass`.

Definition of a lexicon and a grammar accounts to defining these six elements making use of the set of 30 interface functions, namely: `cat`, `lng`, `rng`, `preds`, `succs`, `ldp`, `rdp`, `dp`, `lhd`, `rhd`, `hd`, `ldpr`, `rdpr`, `dpr`, `lhdr`, `rhdr`, `hdr`, `ldpBy`, `rdpBy`, `dpBy`, `lhdBy`, `rhdBy`, `hdBy`, `lm`, `rm`, `hdless`, `clo`, `rclo`, `mclo`, `mrclo` supplemented with ... the whole Haskell environment. Two examples are given below. It should be stressed that the examples are by no means meant to be understood as proposals of grammatical systems or descriptive solutions, they unique role is the illustration of using Haskell language for the purpose of formulating grammatical description.

5.1 Example 1

The first example is minimalistic. We will implement a free word order grammar which is able to analyze Latin sentences composed of words *Joannes*, *Mariam*, *amat*. The six elements required by the parser are presented below. The part of speech affixes 'n' and 'a' stand for 'nominative' and 'accusative'.

```
data Cat  = Nn | Na | V deriving (Eq,Ord)
data Role = Subj | Cmpl deriving (Eq,Ord)

dic "Joannes"   = [Nn]
dic "Mariam"    = [Na]
dic "amat"      = [V]

heads d = [ (r,h) | h ← preds d,
                    r ← link (cat h) (cat d) ]

deps  h = [ (r,d) | d ← preds h,
                    r ← link (cat h) (cat d) ]

pass    = const True

link V Nn       = [Subj]
link V Na       = [Cmpl]
link _ _        = []
```

There is one little problem with the above grammar: duplicate parses are created as a result of attaching the same dependents in different order. we can solve this problem by slightly complicating the definition of `deps` function and substituting the expression `lm ∘ (ldp ⊕ pure) >=> preds` in the place of `preds`. This expression defines a function which returns predecessors (`preds`) of the leftmost (`lm`) left dependent (`ldp`) of the argument node or of the node itself (`pure`) if no dependents are present yet.
Examples of the parser's output:

```
> parse "Joannes amat Mariam"

[ [ Step 3 Nacc [(Cmpl,2)] [],
    Step 2 V [] [(Subj,1)],
    Step 1 Nnom [] [] ] ]

> parse "Joannes Mariam amat"

[ [ Step 3 V [] [(Subj,1),(Cmpl,2)],
    Step 2 Nacc [] [],
    Step 1 Nnom [] [] ] ]
```

The parsing algorithm which results from combining the parser from Section 4 with the above grammar is basically equivalent to the ESDU variant from (Covington, 2001).

5.2 Example 2

The second example shows a more expressive grammar architecture which allows for handling some complex linguistic phenomena such as: constraints on cardinality of roles in dependent connections; local[5] agreement; non-local agreement between coordinated nouns; non-local requirement of a relative pronoun to be present inside a verb phrase in order to consider it as a relative clause; long distance agreement between a noun and a relative pronoun nested arbitrarily deep in the relative clause.

These phenomena are present for example in Slavonic languages such as Polish. In this example the projectivity requirement will be additionally imposed on the tree structures.

In the set of categories, the case and gender markers are taken into account: `n`=nominative, `a`=accusative, `m`=masculine, `f`=feminine; `REL`=relative pronoun. The lexicon is implemented as before[6]:

```
data Cat  = Nmn | Nfn | Nma | Nfa | Vm | Vf
          | ADJmn | ADJfn | ADJma | ADJfa
          | RELmn | RELfn | RELma | RELfa | CONJ
          deriving (Eq,Ord)

data Role = Subj | Cmpl | Coord | CCmpl | Rel | Mod
          deriving (Eq,Ord)

dic "Jan"     = [Nmn]     dic "widział" = [Vm]
dic "Jana"    = [Nma]     dic "widziała"= [Vf]
dic "Maria"   = [Nfn]     dic "czyta"   = [Vm,Vf]
dic "Marię"   = [Nfa]     dic "czytał"  = [Vm]
dic "książka" = [Nfn]     dic "czytała" = [Vf]
dic "książkę" = [Nfa]     dic "który"   = [RELmn]
dic "dobra"   = [ADJfn]   dic "którego" = [RELma]
dic "dobrą"   = [ADJfa]   dic "która"   = [RELfn]
                          dic "którą"   = [RELfa]
                          dic "i"       = [CONJ]
```

We introduce word classes, which are technically predicates on nodes. Functions of type `a →Bool` are instances of `Lattice` class and may be combined with operators ∨ (join) and ∧ (meet), e.g. `nominal` class:

```
v,n,adj,rel,conj :: Node → Bool
v    = (∈ [Vm,Vf]) ∘ cat
n    = (∈ [Nmn,Nma,Nfn,Nfa]) ∘ cat
adj  = (∈ [ADJmn,ADJma,ADJfn,ADJfa]) ∘ cat
rel  = (∈ [RELmn,RELma,RELfn,RELfa]) ∘ cat
conj = (≡ CONJ) ∘ cat

nominal :: Node → Bool
nominal = n ∨ rel

nom,acc,masc,fem :: Node → Bool
```

[5] By the term *local* we mean: limited to the context of a single dependency connection.

[6] *Jan(a)* = John, *Mari(a/ę)* = Mary, *książk(a/ę)* = book, *dobr(a/ą)* = good, *widział(a)* = to see$_{PAST}$, *czyta*=to read$_{PRES}$, *czytał(a)* = to read$_{PAST}$, *któr(y/ego/a/ą)* = which/who/that, *i* = and

```
nom  = (∈ [Nmn,Nfn,ADJmn,ADJfn,RELmn,RELfn]) ∘ cat
acc  = (∈ [Nma,Nfa,ADJma,ADJfa,RELma,RELfa]) ∘ cat
masc = (∈ [Vm,Nmn,Nma,ADJmn,ADJma,RELmn,RELma]) ∘ cat
fem  = (∈ [Vf,Nfn,Nfa,ADJfn,ADJfa,RELfn,RELfa]) ∘ cat
```

The grammar has the form of a list of rules. The type `Rule` is defined as follows:

```
data Rule = Rule Role (Node→Bool) (Node→Bool) [Constr]
```

A value of type `Rule` is built of the type constructor of the same name and four arguments: the first is the dependency type (role), the next two specify categories allowed for the head and the dependent, given in the form of predicates on nodes. The fourth argument of is the list of constraints imposing additional conditions. The type of a constraint in a function from a pair of nodes (head, dependent) to `Bool`.

```
type Constr = (Node,Node) → Bool
```

The functions `heads`, `deps`, and `pass` take the following form:

```
heads d = [ (r,h) | h ← visible d, r ← roles h d ]
deps  h = [ (r,d) | d ← visible h, r ← roles h d ]
pass    = const True

visible = mrclo (lm ∘ dp) >=> lng >=> rclo lhd

roles h d = [ r | Rule r p q cs ← rules,
                  p h,  q d,
                  all ($ (h,d)) cs ]
```

The function `visible` (see Figure 3) returns the list of nodes connectable without violating the projectivity requirement. These are reflexive transitive left heads (`rclo lhd`) of the left neighbour (`lng`) of the maximal transitive leftmost dependent (`mrclo (lm∘dp)`). The function `roles`, given two nodes as arguments, selects roles which may label dependency connection between them. For each rule `Rule r p q cs` in the list of rules, it checks whether the head and dependent nodes satisfy the predicates imposed on their categories (`p` and `q`, respectively), then verifies whether all constrains `cs` apply to the head-dependent pair (`($ (h,d))`.

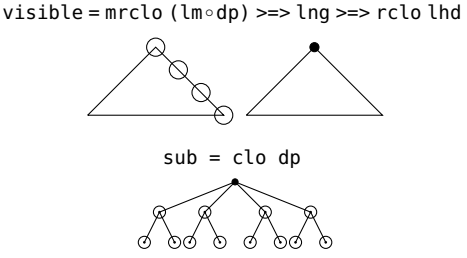

Figure 3: Node functions `visible` and `sub`

The set of constraints for our example include order constraints (`left`, `right`), agreement in gender (`agrG`), case (`agrC`), both case and gender (`agrCG`), agreement between coordinated nouns (`agrCoord`), and the constraint related to relative close attachment (`agrRel`, see below).

```
right    (h,d) = h < d
left     (h,d) = d < h
agrG     (h,d) = (all masc ∨ all fem) [h,d]
agrC     (h,d) = (all nom ∨ all acc) [h,d]
agrCG          = agrC ∧ agrG
agrCoord (h,d) = or [ agrC  (h',d) | h'← hdBy Coord h ]
agrRel   (h,d) = or [ agrCG (h,d') | d'← sub d, rel d']
   where sub = clo dp
```

The constraint `agrCoord`[7] checks whether the node `h` has the head `h'` linked by dependency of type `Coord` and the `agrC` constraint for `h'` and `d` evaluates to `True`; `agrRel` checks whether the node `d` has a transitive dependent `d'` (i.e. subordinate node, function `sub` – see Figure 3) belonging to the category `rel` which agrees with the node `h` in case and gender. Finally, the list of grammar rules may be stated as:

```
rules = [ Rule Subj  v    (nominal ∧ nom)  [agrG],
          Rule Cmpl  v    (nominal ∧ acc)  [],
          Rule Coord n    conj     [right],
          Rule CCmpl conj n        [right,agrCoord],
          Rule Rel   n    v        [agrRel],
          Rule Mod   n    adj      [agrCG]          ]
```

Now, we will extend our grammar with constraints on the cardinality of roles. Let's introduce two more componenents to the grammar: the set of roles, which may appear at most once for each head (`sgl`) and the statements indicating roles which are obligatory for word categories (`obl`).

```
sgl :: [ Role ]
sgl = [ Subj, Cmpl, CCmpl, Rel ]

obl :: [ ((Node → Bool), [Role]) ]
obl = [ (conj,[CCmpl]) ]
```

Singleness constraint will be defined as an instance of a more general mechanism: universal constraints – similar to constraints in rules but with global scope.

```
type UConstr = (Role,Node,Node) → Bool

singleness :: UConstr
singleness (r,h,d) = ¬ (r ∈ sgl ∧ r ∈ dpr h)

uc :: [UConstr]
uc = [singleness]
```

[7] We used the standard Haskell function `or` here, despite its name is not intuitively fitting the context, because it does exactly what we need: it checks both whether the constraint `agrC` returns `True` and whether there esists `h'` for which the `agrC` is evaluated.

Universal constraints will be checked before each dependency is added and will block the addition in case any of them is violated. In order to incorporate them into our grammar we have to replace the function `roles` used in the definition of `heads` and `deps` functions with `roles'` defined as follows:

```
roles' h d = [ r | r ← roles h d, all ($ (r,h,d)) uc ]
```

The function `roles'` extends `roles` by additionally checking if all universal constraints (the list `uc`) apply to the connection being under consideration.

The obligatoriness constraint will be checked after completing the parse, in the `pass` function. The `sat` function looks for all roles which are obligatory for the argument node, as defined in the statements in the `obl` list, and verifies if all of them are present.

```
sat n = all (∈ dpr n) [ r | (p,rs) ← obl, p n, r ← rs ]

pass = all sat ∘ (pure ⊕ preds)   (redefinition)
```

Here are some examples of the parser's output:

```
> parse "widział Marię i Jana"[8]

[ [ Step 4 Nma  [(CCmpl,3)] [],
    Step 3 CONJ [(Conj,2)]  [],
    Step 2 Nfa  [(Cmpl,1)]  [],
    Step 1 Vm   []          []  ]
]


> parse "widział Marię i Jan"[9]

[ ]


> parse "Jan widział książkę którą czyta Maria"[10]

[ [ Step 6 Nfn   [(Subj,5)] [],
    Step 5 Vf    [(Rel,3)]  [(Cmpl,4)],
    Step 4 RELfa []         [],
    Step 3 Nfa   [(Cmpl,2)] [],
    Step 2 Vm    []         [(Subj,1)],
    Step 1 Nmn   []         []          ]
]


> parse "Jan widział książkę którego czyta Maria"[11]

[ ]
```

[8] *he-saw Mary*+acc *and John*+acc

[9] *he-saw Mary*+acc *and John*+nom (`agrCoord` constraint violated)

[10] *John saw the-book*+fem+acc *which*+fem+acc *Mary is-reading*

[11] *John saw the-book*+fem+acc *which*+masc+acc *Mary is-reading* (`agrRel` constraint violated)

6 Efficiency issues

As far as the efficiency issues are concerned, the most important problem appears to be the the number of alternative partial parses built, because partial parses with all possible combinations of legal connections (as well subsets thereof) are produced during the analysis. This may result in unacceptable analysis times for longer and highly ambiguous sentences.

This problem may be overcome by rejecting unpromising partial parses as soon as possible. One of the most obvious selection criteria is the forest size (number of trees in a parse). The relevant parser modification accounts to introducing the selection function (for simplicity we use the fixed value of 4 for the forest size to avoid introducing extra parameters) and redefining the `step` function appropriately:

```
select :: Parse → [Parse]
select p = if size p ≤ 4 then [p] else []

step w = shift w >=> connect >=> select
```

The function `size` which used to compute the number of trees in a parse may be defined as follows:

```
size :: Parse -> Int
size = foldr acc 0
  where acc (Step _ _ h ds) n = n + 1 - length (h ++ ds)
```

7 All/first/best parse variants

The parser is designed to compute all possible parses. If, however, only the first n parses are requested, then only these ones will be computed. Moreover, thanks to the lazy evaluation strategy, only those computations which are necessary to produce the first n parses will be performed. Thus, no modifications are needed to turn the parser into a variant that searches only for the first or first n parses. It is enough to request only the first n parses in the parser invocation. For example, the `parse1` function defined below will compute only the first parse.

```
parse1 = take 1 ∘ parse
```

In order to modify the algorithm to always select the best alternatives according to `someScoringFunction`, instead of the first ones, the parser may by further modified as follows:

```
someScoringFunction :: (Ord a) ⇒ Parse → a
someScoringFunction = ...
```

```
sort :: [Parse] → [Parse]
sort = sortWith someScoringFunction

step w = shift w >=> connect >=> (sort ∘ select)
```

8 Conclusion

In the paper we have tried to show that a lazy functional language such as Haskell is a convenient framework not only for implementing dependency parsers but also for expressing dependency grammars directly as Haskell code. Even without introducing any special notation, language constructs, or additional operators, Haskell itself allows to express the grammar in compact, readable and mathematically clean manner.

The borderline between the parser and the grammar is shifted compared to the traditional view, e.g. CFG/Earley. In the part, which we called the parser core, minimal assumptions are made about structural properties of the syntactic trees allowed (e.g. projective, nonprojective) and the nature of grammatical constraints which are formulated. In fact the only hard-coded requirements are that the syntactic structure is represented in the form of a dependency tree and that the parse is built incrementally.

In order to turn the ideas presented above into a useful NLP tool for building grammars it would be obviously necessary to rewrite the code in more general, parametrizable form, abstracting over word category type (e.g. to allow structural tags), role type, parse filtering and ranking functions, the monad used to represent alternatives, allowing for representing some kinds of weights or ranks etc.

In fact, the work in exactly this direction is already in advanced stage of development. In this paper it was reduced to the essential part (without parameterized data types, multi-parameter classes, monad transformers, and so on), which size allows to present it in full detail and with the complete source code in a conference paper.

References

Covington, M. A. (2001). A fundamental algorithm for dependency parsing. In *In Proceedings of the 39th Annual ACM Southeast Conference*, pages 95–102.

de Kok, D. and Brouwer, H. (2009). Natural language processing for the working programmer. `http://freecomputerbooks.com/books/nlpwp.pdf`. Accessed: 2016-11-22.

Eijck, J. V. (2005). Deductive parsing with sequentially indexed grammars. `http://homepages.cwi.nl/~jve/papers/05/sig/DPS.pdf`. Accessed: 2016-11-22.

Erwig, M. (2001). Inductive graphs and functional graph algorithms. *J. Functional Programming*, 11(5):467–492.

Goodman, J. (1999). Semiring parsing. *Computational Linguistics*, 25(4):573–605.

Hutton, G. and Meijer, E. (1998). Monadic parsing in haskell. *J. Funct. Program.*, 8(4):437–444.

Jones, S. P., editor (2002). *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/.

Koster, C. H. (1992). Affix grammars for natural languages. In Alblas, H. and Melichar, B., editors, *Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*, pages 469–484. Springer-Verlag.

Kubler, S., McDonald, R., Nivre, J., and Hirst, G. (2009). *Dependency Parsing*. Morgan and Claypool Publishers.

Levy, R. and Pollard, C. (2001). Coordination and neutralization in hpsg. In Eynde, F. V., Hellan, L., and Beermann, D., editors, *Proceedings of the 8th International Conference on Head-Driven Phrase Structure Grammar*, pages 221–234. CSLI Publications.

Lipovaca, M. (2011). *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA, 1st edition.

Ljunglöf, P. (2004). Functional chart parsing of context-free grammars. *Journal of Functional Programming*, 14(6):669–680.

Pereira, F. C. N. and Warren, D. H. D. (1980). Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Network. *Artificial Intelligence*, pages 231–278.

Ranta, A. (2011). *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications, Stanford. ISBN-10: 1-57586-626-9 (Paper), 1-57586-627-7 (Cloth).

Yorgey, B. (2009). The typeclassopedia. *The Monad. Reader Issue 13*, pages 17–66.