Automatic Definition Extraction using Parser Combinators

Claudia Borg
Dept. of Artificial Intelligence
University of Malta
claudia.borg@um.edu.mt

Gordon J. Pace
Dept. of Computer Science
University of Malta
gordon.pace@um.edu.mt

Abstract

The automatic extraction of definitions from natural language texts has various applications such as the creation of glossaries and question-answering systems. In this paper we look at the extraction of definitions from non-technical texts using parser combinators in Haskell. We argue that this approach gives a general and compositional way of characterising natural language definitions. The parsers we develop are shown to be highly effective in the identification of definitions. Furthermore, we show how we can also automatically transform these parsers into other formats to be readily available for use within an eLearning system.

Index Terms

Functional programming, definition extraction, natural language processing

1. Introduction

A definition — a term together with a description of its meaning or the concept it refers to — can be particularly useful in the understanding of new terms. The extraction of definitions from natural language texts can be useful for diverse applications, including automatic creation of glossaries for the building of dictionaries and in question answering systems.

We are interested in the use of definition extraction to support eLearning, where definitions can help learners conceptualise new terms and understand new concepts. Learning Objects (LOs) normally contain implicit information in natural language form, which would require a lot of work for the tutor to extract manually. We propose techniques to extract these definitions automatically, to support the tutor who would then be able to refine, rather than create this information from scratch. The main challenge is that the text and definitions are typically non-technical ones — as opposed to scientific, or technical texts in which definitions are typically more structured both in terms of linguistic form and layout.

In our framework we use parser combinators, a technique from functional programming, to enable us to develop definition grammars in an abstract and compositional way, enabling a fast and effective development and testing cycle of the parsers in order to refine them. Furthermore, using this approach, we consume multiple streams of input concurrently, carrying not only the actual text, but also linguistic information extracted through external tools, to identify features indicating definitions.

This work was done in collaboration with an EU-funded FP6 project LT4eL[1]. The project is described in more detail in [MLS07], and aims at enhancing Learning Management Systems by using language technologies and semantic knowledge.

2. Background

Rule-based approaches to definition extraction tend to use a combination of linguistic information and cue phrases to identify definitions. In [KM01] and [SW06] the corpora used are technical texts, where definitions are more likely to be well-structured, and thus easier to identify definitions. Other work attempts definition extraction from eLearning texts [WM07] and [GB07] and the Internet [KPP03]. Non-technical texts tend to contain definitions which are ambiguous, uncertain or incomplete compared to technical texts.

In eLearning, LOs are generally created by tutors in different text formats. A corpus of LOs, gathered within the LT4eL project, has been converted to XML with additional linguistic information, and

1. Language Technologies for eLearning `www.lt4el.eu`

manual tagging of over 450 definitions. Furthermore, the definitions have been separated in three different categories to support the identification process:

1) Definitions containing the verb 'to be' as a connector.
   E.g.: 'A joystick is a small lever used mostly in computer games.'
2) Definitions containing other verbs as connectors such as 'means', 'is defined' or 'is referred to as'.
   E.g.: 'the ability to copy any text fragment and to move it as a solid object anywhere within a text, or to another text, usually referred to as cut-and-paste.'
3) Definitions containing punctuation features separating the term being defined and the definition itself.
   E.g.: 'hardware (the term applied to computers and all the connecting devices like scanners, modems, telephones, and satellites that are tools for information processing and communicating across the globe).'

The approach within LT4eL to identify definitions was through manual observation of grammatical patterns, based on the linguistic information available in the annotated texts. However, this proved to be a tedious task, and required expert knowledge. Our approach makes it easier, even for non-linguist, to experiment with and develop grammars to identify definitions.

3. Parser Combinators for Definition Identification

3.1. The Approach

We propose an infrastructure to define and experiment with parsers embedded inside the general purpose functional programming language Haskell. The system works on textual LOs, and attempts to identify definitions at sentence level. The architecture of our system, as portrayed in figure 1 takes the plain text sentence by sentence, as a stream of words, augmented by streams of linguistic information about the individual words.

The main challenge in using parsers to develop discriminating algorithms for imprecise concepts such as definitions is one of abstraction and effective assessment of the parsers. We use parser combinators, developed in the functional programming community, to describe our definition discriminators for each definition class. Since the parser combinators are built *within* the host language, we develop a whole testing framework to test candidate parsers on manually tagged data. Furthermore, since the parsers are also data structures within the host language, we also develop a number of non-standard[2] interpretations of the parsers. Currently, we can analyse the parsers to assess the use of the different streams, and translate parsers into lxtransduce [Tob05] equivalents. Although not all parsers can be translated directly and automatically, we have found that the parsers we have developed can all be handled by the translator. We envisage the translation of parsers into other formats as the need arises — currently, lxtransduce is used in LT4eL for definition extraction.

3.2. Parser Combinators

Combinator-based programming, where a set of basic objects are defined, together with operators to combine them to produce more complex instances has frequently been used to embed domain-specific languages. One domain in which the combinator approach has been successfully applied is that of parsing. In [HM92] and [Wad85], parser combinators were introduced for Haskell [Jon03], using which, one can compose complex parsers from simpler ones. Furthermore, the core-combinator code for parsing is itself very simple and easy to follow, making it possible to change and add new basic-parsing combinators as other parsing requirements not originally needed arise.

A parser can be seen to be a function which reads an input stream of tokens (of a particular type), and which, after consuming part of the input, returns back a result and the remaining unconsumed part of the input stream. In the case of a non-deterministic parser, a list of such possibilities would be returned. The basic type of a parser which consumes a stream of data of type `a`, and returns a value of type `b`, would thus be the parametrised type `Parser a b`:

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

The list of results enables, not only the possibility of representing non-determinism (by returning multiple results), but also representing failure (by returning the empty list). One advantage of using such a polymorphic type in our case, is that our different streams (words in the sentence, morphological information, part of speech information, etc) can be of different types. Furthermore, we combine these different streams into one stream of tuples of information to enable their consumption in parallel.

Parsers can be easily packaged as monads, which offer a reusable structure for functional programs,

2. The standard interpretation of a parser is considered to be its application to a input stream of text to produce an output.

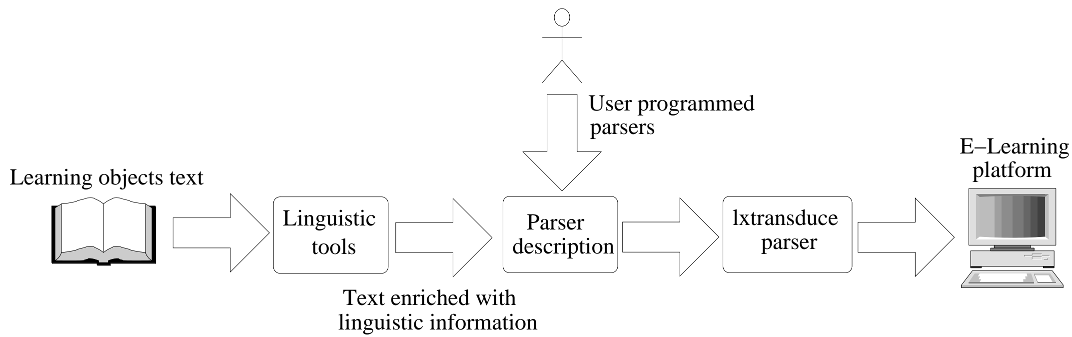

Figure 1. Workflow of definition extraction

enabling the expression of sequentiality in parser combinators [Wad85]. However, for simplicity of presentation, here we use a simpler notation.

Based on this parser type, one can define a number of basic parsers (such as `return x`, which returns value `x` without consuming any of the input stream, and `sat cond`, which consumes one element from the input stream and returns it if it satisfies the given condition `cond`, otherwise fails):

```
return x ys = [(x,ys)]

sat cond [] = []
sat cond (y:ys)
  | cond y    = [(y,ys)]
  | otherwise = []
```

Based on such basic parser combinators which enable combining of a number of parsers into more complex ones (such as |> which composes two parsers in sequence, <+> which composes two parsers to match with either of the two, and <&> which composes two parsers in conjunction). For instance, the code for the sequential composition of two parsers applies the first parser, and then, *for each possible successful parse,* applies the second parser on the remaining input stream, concatenating all the results, and returning the pair of results returned by the two parsers. Note that non-determinism is thus automatically handled within the composition operator:

```
(parser1 |> parser2) ys =
    [ ((x,y), ys'')
    | (x,ys')  <- parser1 ys
    , (y,ys'') <- parser2 ys'
    ]
```

Non-determinism is implemented simply by catenating the possibilities of the parsers:

```
(parser1 <+> parser2) ys =
    parser1 ys ++ parser2 ys
```

Based on these parsers, we can define more complex parsers, such as, for example, `star` which accepts any number of repetitions of a given parser:

```
star parser =
  (parser |> star parser) <+> return ()
```

The last combinator we will use in this paper is the conjunction, or parallel parsing of two parsers — defined to succeed if and only if both parsers succeed on the given input, without returning any concrete value and without consuming any input:

```
(parser1 <&> parser2) ys
  | null (parser1 ys) ||
    null (parser2 ys)    = []
  | otherwise            = [((), ys)]
```

3.3. Definition Parsers

Using these basic parsers and parser combinators, we can define a number of basic parsers for our domain. The input will be a stream of quadruples, consisting of (i) the actual word; (ii) morphological information; (iii) part-of-speech information; and (iv) the lemma of the word. One can thus define basic parsers such as `isNoun` and `posIs` using the `sat` parser:

```
posIs x = sat (\(_,_,pos,_) -> pos == x)
isNoun  = posIs "NN"
```

Note that the condition passed as a parameter to `sat` is a lambda expression which returns the third item in the four-tuples in the input streams. Based on these underlying parsers, one can define grammatical structure parsers. The following example shows how simple noun phrases may be parsed:

```
nounPhrase =
  ( isDeterminer |>
    star isAdjective |>
    isNoun
  ) <+>
  ( star isAdjective |>
    (isPluralNoun <+> isProperNoun)
```

```
)
```

Finally, on the basis of these linguistic structures, we define classes of definitions. The following example is a parser for the definition of a noun using an ‘is a’ definition:

```
nounDefinition =
  star isNotVerb |>
  isNoun |>
  star isNotVerb |>
  isToBe |>
  star isAdverb |>
  nounPhrase
```

These definitions are refined using extra checks on the structure and words in the sentence being parsed:

```
definition_IsA
    =  posIsNot "wrb"
   <&> doesNotInclude (lemmaIs "example")
   <&> doesNotInclude (wordIs "not")
   <&> (verbDefinition <+> nounDefinition)
```

3.4. Non-Standard Interpretations

As can be seen in the examples we have given, the combinators are used to compositionally describe a definition parser progressively. The parsers are simply functions which can be applied to input streams, consuming the data, and returning results. However, it is desirable to be able to process parsers in alternative ways for other purposes. Although functions are first-class objects in functional languages, thus enabling higher order descriptions, they cannot be compared or pattern-matched. We thus overload the parser combinators so as to allow other interpretations of the parser, to enable the creation of a structure which can be analysed and traversed. Currently, we have two alternative interpretations — an analysis to assess dependency on different linguistic information (the constituent parts of the tuples coming in on the stream), and an automatic translation of a given parser into lxtransduce [Tob05] input. The latter is used to link our parser to the eLearning tool.

4. Evaluation

The table below shows our results — a total of 16,042 sentences from non-technical LOs were used, with 79 is-a definitions, 133 verb definitions and 126 punctuation definitions. The remaining sentences are all non-definitions.

Category	Precision	Recall	F-Measure
is-a	32%	86%	46%
verb	40%	40%	40%
punct	25%	52%	34%
Overall	31%	55%	39%

Our results are comparable to other work mentioned in this area. We obtain higher precision and recall for the is-a category than [GB07], [WM07] and a much higher precision than [GB07] (12%) but lower recall (73%) for the verb category. [WM07] obtain better results in the latter category (precision = 45%; recall = 76%). In punctuation category, we manage to obtain a much better precision than [WM07] with a lower recall (precision = 10%; recall = 68%). In [SW06] the corpus used consists of technical texts, and only the first two categories are considered. Considering that technical texts are generally more well-structured, our overall results over these two categories (precision 35%, recall 57%) compares favourably to theirs (precision 34%, recall 70%).

The main problem in the domain of definition extraction is that there are far more non-definitions than definitions. In the case of eLearning, the question that we have to ask is what is the ratio of definitions vs. non-definitions that a tutor is willing to be prompted with to make the tool of definition extraction usable. The balance between recall and precision is quite subjective and different requirements would be set by both the domain of the LOs and by the tutors who use the tools.

5. Conclusions

We have outlined a definition extraction system using parser combinators in Haskell. We develop effective parsers, which can be plugged in directly into an eLearning system, to support glossary creation and question-answering with minimal additional work to be performed by the tutor.

Although the definition extraction is only performed once on a repository of LOs, these can be very large and thus it is important to address efficiency issues. Since we apply the definition parsers at a sentence level, non-determinism fanout is not a big problem. Furthermore, since we seek only one match for the parser, the lazy reduction strategy used in Haskell ensures that not all possible parser matchings are generated. In practice, once the linguistic information is extracted, large documents can be analysed in a fraction of a second.

There are various issues we plan to address in the future. We are currently looking into the use of

statistical, and machine learning techniques to aid the development of grammars. We are also looking into the use of additional linguistic information, such as phrase chunking, with parsing now being performed in an asynchronous manner across streams. These techniques present new challenges which still need to be addressed. However, we strongly believe that the combinator based approach can help hide away all these intricacies within the basic combinators, thus leaving the parsers essentially unchanged.

References

[GB07] Rosa Del Gaudio and António Branco. Automatic Extraction of Definitions in Portuguese: A Rule-based Approach. In *RANLP workshop: Natural Language Processing and Knowledge Representation for eLearning Environments*, 2007.

[HM92] G. Hutton and E. Meijer. Monadic parser combinators. *Journal of Functional Programming*, 2(3):323–343, 1992.

[Jon03] Simon Peyton Jones. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003.

[KM01] Judith Klavans and Smaranda Muresan. Evaluation of the DEFINDER System for Fully Automatic Glossary Construction. In *Proceedings of the American Medical Informatics Association Symposium (AMIA)*, 2001.

[KPP03] Judith L. Klavans, Samuel Popper, and Rebecca Passonneau. Tackling the internet glossary glut: Automatic extraction and evaluation of genus phrases. In *SIGIR'03 Workshop on Semantic Web*, 2003.

[MLS07] Paola Monachesi, Lothar Lemnitzer, and Kiril Simov. Language Technology for eLearning. In *First European Conference on Technology Enhanced Learning*, 2007.

[SW06] Angelika Storrer and Sandra Wellinghoff. Automated detection and annotation of term definitions in german text corpora. In *LREC*, 2006.

[Tob05] Richard Tobin. Lxtransduce A replacement for fsgmatch. Technical report, University of Edinburgh, 2005.

[Wad85] P. Wadler. How to replace a failure by a list of successes. *FPCA*, 201:113–128, 1985.

[WM07] Eline Westerhout and Paola Monachesi. Extracting of Dutch Definitory Contexts for elearning purposes. In *CLIN 2007*, 2007.

View publication stats