Using Types to Parse Natural Language

Mark P. Jones
University of Nottingham
Nottingham, England

Paul Hudak
Sebastian Shaumyan
Yale University
New Haven, Connecticut, USA

Abstract

We describe a natural language parser that uses type information to determine the grammatical structure of simple sentences and phrases. This stands in contrast to studies of type inference where types and grammatical structure play opposite roles, the former being determined by the latter. Our parser is implemented in Haskell and is based on a linguistic theory called *applicative universal grammar* (AUG). Our results should be interesting to computer scientists in the way in which AUG relates to types and combinatory calculus, and to linguists in the way in which a very simple, brute force parsing strategy performs surprisingly well in both performance and accuracy.

1 Introduction

The study of type inference for functional languages depends on the ability to use the syntactic structure of a program term to determine the types of its components. Such languages are specified by simple context free grammars that provide strong hints about syntactic structure using explicit punctuation such as the 'λ' and '.' symbols in a λ-term, or parentheses to express grouping. As a result, it is easy to parse a flat program text and to construct the corresponding term.

Parsing natural language text is much more difficult. One reason is that grammars for natural languages are often complex, ambiguous, and specified by collections of examples rather than complete formal rules. Another difficulty is that punctuation is used much more sparingly. For example, many sentences in English consist of a sequence of words in which the only punctuation is the terminating period.

In this paper, we describe a program, written in the functional language Haskell, for parsing natural language phrases using type information to determine the required grammatical structure. This stands in contrast to the description of type inference above where these roles are reversed, structure determining type.

Natural language processing is of course a very rich and diverse research area, and space limitations preclude a summary of techniques. However, the topic of natural language processing in a functional language has also been discussed by Frost and Launchbury [5]. Their work differs from ours by its foundation on a semantic theory that is based on principles proposed by Montague [12]. The Frost and Launchbury system includes a parser, implemented using standard combinator parsing techniques [9], and, unlike the program described in this paper, a simple, interactive query system. On the other hand, their approach seems limited by the fact that the grammar for natural language phrases is fixed as part of the parser, and tightly coupled to the underlying semantic model.

1.1 Applicative Universal Grammar

Our work is based on the formalism of *applicative universal grammar* (AUG), a linguistic theory that views the formation of phrases in a form that is analogous to function application in a programming language. The first complete description of AUG was published in 1965 [13], unifying the categorial calculus of Lesniewski [10] with the combinatory calculus of Curry and Feys [4]. The semantic theory of AUG was presented in [14], and its use in the translation of natural languages is given in [16]. A full description of the current state of AUG is described in [15].

To understand the way that AUG works, it is useful to think of words and phrases as atoms and expressions, respectively, in a typed language of combinators. For our simplified version of AUG, there are just two primitive types: `T` representing terms (for example, nouns such as 'friend' and noun phrases such as 'my friend'), and `S` representing complete sentences (such as 'my friend runs'). The only non-primitive type is of the form $\mathtt{O}xy$, denoting phrases that transform phrases of type x to modified phrases of type y; this is the most important concept behind the AUG formalism.

For example, the word 'my' is treated as having type `OTT` since it is applied to a term of type `T` to obtain a modified term, also of type `T` (every word is pre-assigned one or more types in this way). Thus the construction of the noun phrase 'my friend' can be described by an inference:

$$\frac{\text{`my'} :: \mathtt{OTT} \quad \text{`friend'} :: \mathtt{T}}{\text{`my friend'} :: \mathtt{T}}$$

More generally, we can use the following rule to describe the application of one phrase, p of type $\mathtt{O}xy$, to another, q of type x:

$$\frac{p :: \mathtt{O}xy \quad q :: x}{p\, q :: y}$$

Clearly, types of the form $\mathtt{O}xy$ correspond to function types, written as $(x \to y)$ in more conventional notation, while the typing rule above is the standard method for typing the application of a function p to an argument value q. The `O` for function types is used in the descriptions of AUG cited above, and for the most part we will continue to use the same notation here to avoid any confusion with type expressions in Haskell; in our program, the types of natural language phrases are represented by data values, not by Haskell types. Another advantage of the prefix `O` notation is that it avoids the need for parentheses and allows a more compact notation for types.

The results of parsing a complete sentence can be described by a tree structure labelled with the types of the words and phrases that are used in its construction. The following example is produced directly by the program described later from the input string `"my friend lives in Boston"`.

```
                         in      Boston
                    [OTOOTSOTS]   [T]
  my   friend lives      \________/
[OTT]   [T]   [OTS]       [OOTSOTS]
  \_____/      \_____________/
    [T]            [OTS]
     \________________/
             [S]
```

Notice that, to maintain the original word order, we have allowed both forward and backward application of functions to arguments. The first of these was described by the rule above, while the second is just:

$$\frac{q :: x \quad p :: \mathtt{O}xy}{q\,p :: y}$$

For example, in the tree above, we have used this rule to apply the phrase `in Boston` to the intransitive verb `lives`; the function acts as a modifier, turning the action of 'living' into the more specific action of 'living in Boston'.

It is sometimes useful to rearrange the trees produced by parsing a phrase so that functions are always written to the left of the arguments to which they are applied. This reveals the *applicative* structure of a particular phrase and helps us to concentrate on underlying grammatical structure without being distracted by concerns about word order — which vary considerably from one language to another. Rewriting the parse tree above in this way we obtain:

```
     in      Boston
[OTOOTSOTS]   [T]
     \________/    lives   my   friend
      [OOTSOTS]    [OTS] [OTT]   [T]
          \__________/     \_____/
             [OTS]           [T]
               \_____________/
                     [S]
```

In situations where the types of subphrases are not required, we can use a flattened, curried form of these trees, such as `in Boston lives (my friend)`, to describe the result of parsing a phrase. The two different ways of arranging a parse tree shown here correspond to the concepts of *phenotype* and *genotype* grammar, respectively, in AUG, but will not be discussed in any further detail here.

One of the most important tasks in an application of AUG is to assign suitable types to each word in some given lexicon or dictionary. The type `T` is an obvious choice for simple nouns like 'friend' and 'Boston' in the example above. Possessive pronouns like 'my' can be treated in the same way as adjectives using the type `OTT`. In a similar way, intransitive verbs, like 'lives', can be described by the type `OTS` transforming a subject term of type `T` into a sentence phrase of type `S`. The word 'in', with type `OTOOTSOTS`, in the example above deserves special attention. Motivated by the diagram above, we can think of 'in' as a function that combines a place of type `T` (where?), an action of type `OTS` (what?), and a subject of type `T` (who?) to obtain a sentence phrase of type `S`.

One additional complication we will need to deal with is that, in the general case, a single word may be used in several different ways, with a different type for each. In this paper we adopt a simple solution to this problem by storing a list of types for each word in the lexicon. We will see later how we can take advantage of this, including the possibility of a word having several roles (and types) *simultaneously* in the same sentence.

1.2 Functional Programming in Haskell

In contrast to the references above, most of which are aimed at those with a background in linguistics, this paper is intended to be read by computer scientists and, in particular, those with an interest in functional programming. The programs in this paper are written using Haskell [7], a standard for non-strict purely functional programming languages. Tutorial information on these languages may be found elsewhere [1, 6]. Our use of Haskell is fitting since the language is, in fact, named for the logician Haskell B. Curry whose work on combinatory logic cited above provides much of the foundation for both functional programming and AUG. Indeed, Curry himself was interested in the study of natural language and grammatical structure [3].

The LaTeX source for this paper is also a literate script for the program that it describes. In other words, the same file used to produce the document that you are now reading also serves as the source code for the program that it describes.[1] Program lines are distinguished by a '>' character in the first column. The source file also contains some small sections of code that are used to print ASCII versions of tree structures (as illustrated by the example above), and to implement a dictionary assigning types to a small vocabulary of words. These items are not shown in the typeset version of the paper, in an attempt to avoid unnecessary distraction from our main subject. Full source code is available from the authors.

2 Types, Trees and Sentences

Our first task in the implementation of the parser is to choose a representation for types. Motivated by the description above, we define:

```
> data Type = T | S | O Type Type   deriving Eq
```

The specification `deriving Eq` declares that the new datatype `Type` is a member of Haskell's pre-defined class `Eq`, and that the system should therefore derive a definition of equality on values of type `Type`. This is needed so that we can test that the argument type of a function is equal to the type of value that it is applied to. We also include `Type` in the standard Haskell class `Text` so that `Type` values can be displayed using the notation described earlier, without parentheses or whitespace.

[1] This "literate programming style" was originally promoted by Donald Knuth.

The result of parsing a string will be a tree structure with each node annotated with a list of types (each type corresponding to one possible parse).

```
> type TTree = (Tree,[Type])
> data Tree = Atom String | FAp TTree TTree | BAp TTree TTree
```

Applications of one tree structure to another are represented using the `FAp` (forward application) and `BAp` (backward application) constructors.

We will also need methods for displaying typed tree structures. To display the applicative structure of a tree value without the type annotations, we extend the `Text` class with an instance definition for `Tree` values. We will also use a function:

```
> drawTTree :: TTree -> String
```

to display a typed tree in the form shown in Section 1.1. We do not include the code for these functions here since they are needed only to display output results.

The first step in the parser is to convert an input string into a list of words, each annotated with a list of types. For simplicity, we use the `Atom` constructor so that input sentences can be treated directly as lists of typed trees:

```
> type Sentence = [TTree]

> sentence :: String -> Sentence
> sentence = map wordToTTree . words
>  where wordToTTree w = (Atom w, wordTypes w)
```

The function `wordTypes` used here maps individual words to the corresponding list of types. For example, `wordTypes "friend" = [T]`. This function can be implemented in several different ways, for example, using an association list or, for faster lookup, a binary search tree. For all of the examples in this paper, we used a simple (unbalanced) binary search tree containing 62 words. However, we will not concern ourselves with any further details of the implementation of `wordTypes` here.

The following text strings will be used to illustrate the use of the parser in following sections:

```
> myfriend  = "my friend lives in Boston"
> oldfriend = "my old friend who comes from Moscow"
> long      = "my old friend who comes from Moscow thinks that\
>            \ the film which he saw today was very interesting"
```

For example, the first stage in parsing the `myfriend` string is to split it into the following list of typed tree values:

```
? sentence myfriend
[(Atom "my",[OTT]),
 (Atom "friend",[T]),
 (Atom "lives",[OTS]),
 (Atom "in",[OTOOTSOTS]),
 (Atom "Boston",[T])]
```

3 From Sentences to Trees

We have already described how individual words, or more generally, phrases can be combined by applying one to another. Now consider the task of parsing a sentence consisting of a list of words `[`w_1`, ..., `w_n`]`. One way to proceed would be to choose a pair of adjacent words, w_i and w_{i+1}, and replace them with the single compound phrase formed by applying one to the other, assuming, of course, that the types are compatible. Repeating this process a total of $n-1$ times reduces the original list to a singleton containing a parse of the given sentence.

The most important aspect of this process is not the order in which pairs of phrases are combined, but rather the tree structure of the final parsed terms. In this sense, the goal of the parser is to find all well-typed tree structures that can be formed by combining adjacent phrases taken from a given list of words.

3.1 Enumerating Types/Trees

We wish to define the following function to enumerate all of the typed trees that can be obtained from a given sentence:

```
> ttrees :: Sentence -> [TTree]
```

The simplest case is when the list has just one element, and hence there is just one possible type:

```
> ttrees [t] = [t]
```

For the remaining case, suppose that we split the input list `ts` into two non-empty lists `ls`, `rs` such that `ts = ls ++ rs`. Using recursion, we can find all the trees `l` than can be obtained from `ls` and all the trees `r` that can be obtained from `rs`. We then wish to consider all pairs of these that can be combined properly to form a well-typed phrase. This yields the final line in the definition of `ttrees`:

```
> ttrees ts  = [ t | (ls,rs) <- splits ts, l <- ttrees ls,
>                                          r <- ttrees rs,
>                                          t <- combine l r ]
```

The function `splits` is used here to generate all pairs of non-empty lists `(ls,rs)` such that `ls ++ rs = ts`. It can be defined using:

```
> splits       :: [a] -> [([a],[a])]
> splits ts     = zip (inits ts) (tails ts)

> inits, tails :: [a] -> [[a]]
> inits [x]     = []
> inits (x:xs)  = map (x:) ([]:inits xs)

> tails [x]     = []
> tails (x:xs)  = xs : tails xs
```

For example:

```
? inits "abcde"
["a", "ab", "abc", "abcd"]
? tails "abcdef"
["bcde", "cde", "de", "e"]
? splits "abcdef"
[("a","bcde"), ("ab","cde"), ("abc","de"), ("abcd","e")]
```

The function `combine` is used in `ttrees` to generate all possible typed trees, if any, that can be obtained by combining two given typed trees. For the framework used in this paper, the only way that we can combine these terms is to apply one to the other.[2] To allow for variations in word order, we consider both the possibility that `l` is applied to `r`, and also that `r` is applied to `l`:

```
> combine     :: TTree -> TTree -> [TTree]
> combine l r = app FAp l r ++ app BAp r l
```

The rule for application of one term to another is encoded as follows:

```
> app :: (TTree -> TTree -> Tree) -> TTree -> TTree -> [TTree]
> app op (a,ts) (b,ss)
>   = [ (op (a,[O x y]) (b,[x]), [y]) | (O x y)<-ts, z<-ss, x==z ]
```

The expression `(op (a,[O x y]) (b,[x]), [y])` here corresponds to the rule that, if `a` has type `O x y` and `b` has type `x`, then the application of `a` to `b` has type `y`. The use of singleton lists signals that the type of an application is uniquely determined by the type of its arguments. Clearly, we could extend the definition of `combine` to deal with other methods of combining terms in extended AUG frameworks.

The fact that we allow two different ways of combining a pair of terms by applying either one to the other, causes an exponential increase in the number of possible parse trees that might, in theory, need to be considered. For example, we can show that there are 8,448 different ways to construct a parse tree for a sentence like `oldfriend` in Section 2 with only 7 words! Fortunately, the use of types eliminates almost all of these. Using the Gofer interpreter, we obtain just three parses for this sentence with no noticeable delay:

```
? (map show . ttrees . sentence) oldfriend
["(my (old (who friend (from Moscow comes))),[T])",
 "(my (who (old friend) (from Moscow comes)),[T])",
 "(who (my (old friend)) (from Moscow comes),[T])"]
(8302 reductions, 23220 cells)
```

We comment on these parses in more detail in Section 4.

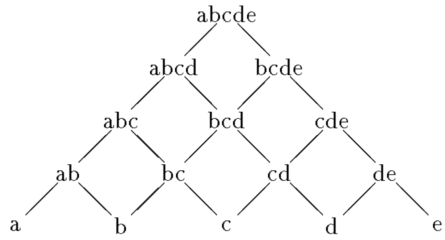
Unfortunately, for larger sentences, the definition of `ttrees` is not efficient enough. For example, evaluation of `(map show . ttrees . sentence) long`, where `long` is the 19 word sentence defined in Section 2, takes 644,776,714

[2] This limitation is not as severe as it might sound, linguistically, since *currying* permits application to several arguments. The parse described earlier in Section 1.1 involving the word 'in', with type `OTOOTSOTS`, is an example of this, as are transitive and ditransitive verbs, having types `OTOTS` and `OTOTOTS`, respectively.

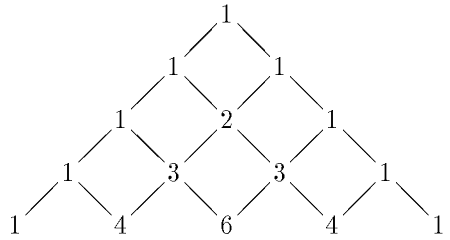
reductions, 1,725,184,431 cells and runs for over 8 hours on a conventional workstation. In total, the program produces 60 different parses, compared with a theoretical maximum of 926,554,883,358,720. These figures are astronomical! Given a search space this large, it is quite an achievement to have constructed an algorithm that terminates within a few hours. However, with this kind of performance, the `ttrees` function is clearly unsuitable for practical natural language processing applications.

3.2 A More Sophisticated Algorithm

Fortunately, there is a way to make the definition of `ttrees` more efficient, but first we need to take a closer look at how the function works when it is applied to a typical sentence. For brevity, suppose that we have a sentence with only five words which we can represent by the five character string ‘abcde’. The following diagram illustrates the pattern of recursive calls to find the value of `ttrees` for this string:



The calculation of `ttrees` at any particular point in this diagram requires recursive calls to `ttrees` for each of the points on the sloping lines below it. For example, to calculate `ttrees` at the root node, we need to find the value of the function for each of the values in the pairs (a,bcde), (ab,cde), (abc,de), (abcd,e); these are just the values returned by `splits`. Notice that the recursive calls for bcde and abcd will both require further recursive calls to find the value of `ttrees` at bcd – so this particular computation will be duplicated. The calculations for points further down the tree will be repeated even more times than this, in the familiar pattern of Pascal’s triangle:



For an n word sentence, the total number of recursive calls to `ttrees` can be calculated by summing all of the entries in the first n rows of Pascal’s triangle, amounting to $2^n - 1$ calls (some of which will be more expensive than others). But, in fact, only $1 + \ldots + n = n(n + 1)/2$ different values are required. For the nineteen word sentence considered above, this means that there are a total of 524,827 calls when only 190 different values are needed.

One way to avoid the repeated calculations here would be to use a language in which `ttrees` was defined as a ‘memo-function’ [11, 8]. This would allow the underlying implementation to avoid repeated calculation whenever possible by reusing values from a cache of previously calculated (argument,result) pairs. However, memo-functions are not supported in Haskell, so we will need to do some extra work to construct and use an explicit cache. Our basic strategy will be to implement a function that calculates, not just one list of typed trees, but a whole table of them, represented by a list of lists:

abcde	abcd	abc	ab	a
bcde	bcd	bc	b	
cde	cd	c		
de	d			
e				

The diagram illustrates how cache tables are built up using recursion: If we have already constructed the table for bcde, then we can construct the table for abcde by adding an extra row. The elements of this row can be calculated from right to left using previously calculated entries to the right and in the column below instead of a recursive call. Once we have built the whole table, the result that we want can be extracted from the top-left position. The following definitions show how this is expressed in Haskell:

```
> fastTtrees = head . head . cache

> cache       :: Sentence -> [[[TTree]]]
> cache [x]    = [[[x]]]
> cache (x:xs) = [build x (transpose rs)] ++ rs
>                where rs = cache xs

> build           :: TTree -> [[[TTree]]] -> [[TTree]]
> build a []       = [[a]]
> build a (ts:tss) = g (reverse is) ts : is
>  where is      = build a tss
>        g is ts = [ r | (i,t) <- zip is ts,
>                        ti   <- i,
>                        tt   <- t,
>                        r    <- combine ti tt ]
```

Those familiar with program transformation will recognize the technique of using a table of intermediate results as an example of *tabulation*. We refer the reader to other work in this area for further details, particularly the recent paper by Bird and de Moor [2] which includes a formal argument that can be used to show that `fastTtrees` is equivalent to the original definition of `ttrees`.

Not surprisingly, the `fastTtrees` function offers a dramatic improvement in performance over the definition of `ttrees`, particularly for long sentences.

For example, running the following program in the Gofer interpreter takes just a second to determine that there are 60 different parses of the 19 word sentence `long`:

```
? (length . fastTtrees . sentence) long
60
(22594 reductions, 52042 cells)
```

This should be compared with the 8 hours, and the mind-boggling counts of reductions and cells for the same calculation using `ttrees` that was described at the end of the previous section.

4 A Simple Example

For the purposes of simple experiments, we combine the components of the parser described above by defining the function:

```
> explain :: String -> String
> explain  = unlines . map drawTTree . fastTtrees . sentence
```

For example, consider the phrase 'my old friend who comes from Moscow'. The result of parsing this phrase using our program are shown in Figure 1. As the figure shows, there are three different ways to parse this phrase, each of which produces a term phrase of type `T`. Without any underlying formal semantics for the language, it is difficult to justify any formal statement about these three parses. However, from an informal standpoint, for example, by observing the grouping of words, we can argue that all three of these parses are valid interpretations of the original phrase, each with slightly different meaning and emphasis:

- `my (old (who friend (from Moscow comes)))`: The words 'friend who comes from Moscow' are grouped together; of all my friends who come from Moscow, this phrase refers to the one that is old.
- `my (who (old friend) (from Moscow comes))`: In this case, the emphasis is on the word 'my'; perhaps you also have an old friend who comes from Moscow, but in this phrase, I am referring specifically to my old friend from Moscow.
- `who (my (old friend)) (from Moscow comes)`: A reference to 'my old friend' who comes from Moscow (but doesn't necessarily live there now).

When we started work on the program described in this paper, we were concerned that the rules for constructing parses of sentences were too liberal and that, even for small sentences, we would obtain many different parses, perhaps including some that did not make any sense. From this perspective, it is encouraging to see that there are only three possible parses of the example sentence used here and that all of them have reasonable interpretations. Of course, it is possible that there may be ways of interpreting this phrase that are not included in the list above; these might be dealt with by adding new types for some of the words involved to reflect different usage or meaning. Another possibility is that we might find a phrase with different interpretations

```
? explain "my old friend who comes from Moscow"

my (old (who friend (from Moscow comes))):

                                        from    Moscow
                                    [OTOOTSOTS]  [T]
             friend    who    comes      \________/
              [T]   [OTOOTST] [OTS]       [OOTSOTS]
               \________/       \_____________/
                 [OOTST]             [OTS]
        old         \__________________/
       [OTT]                 [T]
   my    \____________________/
 [OTT]             [T]
   \________________/
           [T]


my (who (old friend) (from Moscow comes)):

        old  friend                     from    Moscow
       [OTT]  [T]                   [OTOOTSOTS]  [T]
         \_____/       who    comes      \________/
           [T]      [OTOOTST] [OTS]       [OOTSOTS]
            \___________/       \_____________/
               [OOTST]               [OTS]
   my             \____________________/
 [OTT]                      [T]
   \_________________________/
               [T]


who (my (old friend)) (from Moscow comes):

        old  friend
       [OTT]  [T]
   my    \_____/                        from    Moscow
 [OTT]     [T]                      [OTOOTSOTS]  [T]
   \________/          who    comes      \________/
       [T]          [OTOOTST] [OTS]       [OOTSOTS]
        \_______________/       \_____________/
             [OOTST]                 [OTS]
                \______________________/
                           [T]
```

Figure 1: Parsing the phrase ‘my old friend who comes from Moscow’.

that cannot be distinguished by their grammatical structure alone, in which case some form of semantic analysis may be needed to resolve any ambiguities.

While it seems reasonable to allow three different parses for the sentence above, we may be a little concerned about the 60 different parses mentioned above for the 19 word sentence that was used as a test in the previous sections. However, it turns out that half of these parse trees include one of the three different trees for 'my old friend who comes from Moscow' as a proper subphrase; this immediately introduces a factor of three into the number of parses that are generated. Similar multiplying factors of two and five can be observed in other parts of the output. Once we have identified these common elements, the results of the parser are much easier to understand.

Clearly, a useful goal for future work will be to modify the parser to detect and localize the effect of such ambiguities. For example, it might be useful to redefine `TTree` as `([Tree],[Type])` and store lists of subphrase parse trees at each node, rather than generating whole parse trees for each different combination subphrase parse trees.

5 Areas for Further Investigation

The work described in this paper is a promising start, but much remains to be done. From the perspective of natural language processing, we believe that the following points will be useful directions for future work:

- **Ambiguity**: Clearly, some mechanisms are required to deal with ambiguity and multiple parses. In some cases, semantic analysis will be needed to detect and eliminate some parses of a sentence.

- **Type inference**: It would be interesting to experiment with the possibility of inferring types for words that are not already in the vocabulary. For example, most English speakers will be able to understand the sentence 'my friend lives in Beeston', even if they have never heard the word 'Beeston' before. From the context in which the word appears, it is fairly easy to guess that it must refer to a place. Of course, it would not be surprising to respond to this sentence by asking a question such as 'Where is Beeston?'.

  In a similar way, even if the word 'Beeston' does not appear in vocabulary that is recognized by our parser, it is clear that the sentence 'my friend lives in Beeston' would make sense if we were to assume that 'Beeston'::`T`, and perhaps follow this assumption with a query to ask for confirmation, and more information about this new word.

  This apparently straightforward idea may prove rather challenging if we also hope to deal with examples like 'my friend is living in New Haven' (where 'New Haven' should be treated as a single compound term, not as if it were some newer version of a place called 'Haven') and also with 'my friend is living in sin' which has altogether different connotations...

- **A more complete treatment of AUG**: The full theory of AUG goes beyond the extensions we have described here. We would also like to investigate the analysis of lexical/morphological structure in more detail and to study the treatment of punctuation, discontinuous constructions, and metarules of priority in the context of our parser.

The ideas described here also suggest some topics for further investigation in the field of programming language design:

- **Parsing extensible languages**: The techniques used here provide an incremental approach to grammar specification. Instead of the monolithic sets of productions that are normally used to formalize programming language syntax, we can use localized rules to describe how individual parts of a language associate with the other objects round them. This may be a more appropriate model for the syntax of extensible languages, and a companion to recent work on the construction of extensible interpreters and denotational semantics

- **Understanding abstract datatypes**: The use of abstract datatypes in a language like Haskell is controlled by typing rules, in much the same way that types control the use of words in the parse trees shown in this paper. It would be useful to study the grammars implied by the signatures of particular abstract datatypes and to use these to establish formal properties about their use. For example, monads are widely used as a means of embedding imperative effects in a functional languages; the ideas described here might be used to justify optimizations, for example, that a value is used in a single-threaded manner, or that it is safe to implement a given primitive using an imperative effect.

Acknowledgements

The work described in this paper was originally motivated by a series of seminars given by Sebastian Shaumyan and Paul Hudak at Yale University in the fall of 1993, on the subject of natural language processing in Haskell. The preparation of this paper was supported in part by a grant from ARPA, contract number N00014-91-J-4043.

References

[1] R. Bird and P. Wadler. *Introduction to functional programming*. Prentice Hall, 1988.

[2] R.S. Bird and O. de Moor. Relational program derivation and context-free language recognition. In A.W. Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, chapter 2, pages 17–35. Prentice-Hall International Series in Computer Science, 1994.

[3] Haskell B. Curry. Some logical aspects of grammatical structure. In *Structure of language and its mathematical aspects*, Providence, Rhode Island, 1961. American Mathematical Society.

[4] Haskell B. Curry and Robert Feys. *Combinatory logic*, volume Volume I. North-Holland Publishing Company, Amsterdam, 1958.

[5] R. Frost and J. Launchbury. Constructing natural language interpreters in a lazy functional language. *The Computer Journal*, 32(2):108–121, April 1989.

[6] P. Hudak and J. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992. Also available as Research Report YALEU/DCS/RR-901, Yale University, Department of Computer Science, April 1992.

[7] P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[8] John Hughes. Lazy memo-functions. In Jouannaud, editor, *Proceedings of the IFIP conference on Functional Programming Languages and Computer Architecture*, pages 129–146, New York, 1985. Springer-Verlag. Lecture Notes in Computer Science, 201.

[9] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3), July 1992.

[10] Stanislaw Lesniewski. Grundzuge eines neuen Systems der Grundlagen der Mathematik. *Fundamenta Mathematicae*, 14:1–81, 1929.

[11] D. Michie. 'Memo' functions and machine learning. *Nature*, April 1968.

[12] Richard Montague. Formal philosophy. In R.H. Thomason, editor, *Selected writings of Richard Montague*. Yale University Press, New Haven, CT, 1974.

[13] Sebastian Shaumyan. *Strukturnaja lingvistika*. Nauka, Moskva, 1965.

[14] Sebastian Shaumyan. *Applicative grammar as a semantic theory of natural language*. University of Chicago Press, 1977.

[15] Sebastian Shaumyan. *A Semiotic Theory of Language*. Indiana University Press, 1987.

[16] Sebastian Shaumyan. Applicative universal grammar as a linguistic framework of the translation model. In *Proceedings of the Fifth International Conference on Symbolic and Logical Computing*, pages 287–320, Dakota State University, Madison, Dakota, 1991.