Linguistic, Philosophical, and Pragmatic Aspects of Type-Directed Natural Language Parsing

Sebastian Shaumyan¹ and Paul Hudak²

¹ Yale University, Department of Computer Science, New Haven CT 06520, USA ² Yale University, Department of Linguistics, New Haven CT 06520, USA

Abstract. We describe how type information can be used to infer grammatical structure. This is in contrast to conventional type inference in programming languages where the roles are reversed, structure determining type. Our work is based on *Applicative Universal Grammar* (AUG), a linguistic theory that views the formation of phrase in a form that is analogous to function application in a programming language. We descibe our overall methodology including its linguistic and philosophical underpinnings.

The natural language parser that we have constructed 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. Both computer scientists and linguists should also be interested in the convergence of the theory of functional programming languages and AUG with regard to their conceptual bases. Both have been motivated by entirely different goals and have developed independently, but they are rooted in a common conceptual system of an applicative calculus.

1 Functional Programming Languages and Applicative Universal Grammar: a Common Paradigm

The central goal of a *theoretical* study of linguistics is to reveal how a human being uses languages to express and communicate thought. The *philosophy* of language is concerned with evaluating how particular theories of language succeed in attaining this central goal. Finally, the *pragmatics* of language concerns itself with the ability to implement a theory, for example in the form of a natural language processing system. Linguists and computer scientists benefit from exchanges concerning theory, philosophy, and pragmatics.

In this paper we touch on all three of these issues. We first argue that the theories of *functional languages* and *Applicative Universal Grammar* (AUG) share a common underlying paradigm. By paradigm we mean a pattern of thought, a system of concepts and assumptions underlying a group of theories, regardless of how they differ from one another in detail. Some of the fundamental ideas of this common paradigm can be traced to the works of Haskell B. Curry on combinatory logic and its philosophical and linguistic implications. Indeed, our argument leads to the realization of a natural language parser written in the functional language Haskell, and based on the theory of AUG.

The first complete description of AUG was published in 1965 [16], unifying the categorical calculus of Lesniewski [12] with the combinatory calculus of Curry and Feys [5]. The semantic theory of AUG was presented in [17], and its use in the translation of natural languages is given in [19]. A full description of the current state of AUG is given in [18].

A description of the feasibility of natural language parsing using AUG was first given in [11] as a literate Haskell [10] program. The functional language Haskell is named in the memory of Haskell B. Curry, the logician whose work on combinatory logic provides much of the foundation for both functional programming languages and AUG. Indeed, Curry himself was interested in the study of natural language and grammatical structure [4]. Related work on using a functional language for NLP may be found in [6], which differs from ours by being based on principles proposed by Montague [15].

2 The Rise of Functional Programming Languages

The earliest programming languages were developed with one goal in mind: to provide a vehicle through which one could control the behavior of computers. This goal seemed reasonable, but quickly ran into two problems: first, it became obvious that what was easy for a machine was not necessarily easy for a human being; and second, as the number of different kinds of machines increased, the need arose for a common language with which to program them all.

Thus from raw object code and primitive assembly languages, a large number of *high-level* programming languages emerged, beginning with FORTRAN in the 1950's. High-level languages are generally divided into two classes: *imperative* languages and *declarative* languages. Imperative languages are lower level—they say a lot about *how* to compute a result—whereas declarative languages are at a higher level—they say most about *what* should be computed, rather than how. *Functional* programming languages are declarative languages whose underlying model of computation is the function (in contrast to the relation that forms the basis for logic programming languages). (See [8] for a more thorough discussion of these issues.)

The key point here is that higher-level languages are less dependent on the details of the underlying hardware. The earliest programming languages depended on the hardware completely, reflecting fairly accurately the structure of the underlying machine. The emergence of high-level imperative languages was a considerable step forward; still, it was only a partial liberation from the machine. Only through the emergence of declarative languages, in particular pure functional languages, did a fairly complete liberation take place.

Indeed, in studying any language—a programming language, a language of mathematics or logic, or a natural language such as English or Russian—we must distinguish two very different kinds of structure: the *physical* structure and the *functional* structure of language. The physical structure of programming languages is a reflection of the physical structure of the computer. The physical structure of natural language is so-called *phrase-structure*, which reflects the action of the human machine; i.e. the organs of speech. Because of the dependence on the human organs of speech, natural language has a *linear* phrase-structure. However, the essential structure of natural language, its functional structure, is independent of the linear phrase-structure.

The parallel between programming languages and natural languages with respect to the distinction of the physical structure and functional structure is clear. Does this distinction apply to the languages of mathematics and logic? We find a partial answer to this question in the illuminating studies of Haskell B. Curry, who distinguished two kinds of structure: the structure of *concatenative* systems in standard formalizations of logic and the *applicative* structure of *ob-systems* in combinatory logic. In Curry's terminology, a concatenative system is a concrete system depending on physical properties of signs; that is, on their properties represented as linear strings. The concatenation operation is an operation of combining signs in their linear order. Here's how Curry contrasts the concatenative structure and the applicative structure:

"In the first place, combinatory logic was formulated as a system in which the formal objects were rather differently conceived than was the case in the standard formalization of logic. The standard procedure at that time was to demand that the formal objects be expressions of some "object language;" this means that they be strings formed from the symbols of that object language by concatenation. In combinatory logic these formal objects, called obs, were wholly unspecified. It was merely postulated that there was a binary operation of application among them, that the obs be constructed from the primitive objects, called atoms, by these operations, and that the construction of an ob be unique. This means that the obs were thought of not as strings of atoms, but as structures like a genealogical tree. [4, pp. 64–65]

From this discussion we adopt the premise that the applicative structure is the essential structure of any language—a programming language, a language of mathematics or logic, or a natural language.

3 The Rise of Applicative Universal Grammar

Applicative Universal Grammar (AUG) develops the ideas of Ferdinand de Saussure about language as a system of signs. It is based on the observation that some fundamental ideas of linguistics and combinatory logic converge. Thus AUG consists of a formal model of natural languages, called *genotype*, that is based on combinatory logic.

The basic concept of AUG is the *applicative structure*, which contrasts sharply with the concept of *phrase structure*, the standard formal system in many linguistic theories. The essential properties of the applicative structure of AUG are characterized by the following principles:

Principle 1 The Transfer Principle.

A grammatical structure or class is independent of its symbolic representation, and so may be transferred from one symbolic device into another without changing its meaning.

For example, subject-object relations can be represented by case markers as in Russian or Latin or by word order as in English. In a Russian sentence consisting of three words—subject, predicate, and direct object—the words can be permuted in six different ways without changing the meaning of the sentence. Grammatical relations are invariant of their symbolic representations.

Principle 2 The Genotype-Phenotype Principle.

Grammatical structures and classes, independent of their symbolic representation, form a universal system of grammatical categories of natural languages, called a genotype. Symbolic representations of grammatical structures and classes of specific natural languages form concrete systems called phenotypes.

This principle is a corollary of the Transfer Principle, and further constrains grammar by imposing the genotype and phenotype levels on it. It calls for a revision of linguistic theories which confound these levels.

In accordance with the Genotype-Phenotype Principle, we must clearly distinguish *rules of grammar* from *laws of grammar*. Rules of grammar are languagespecific; they are part of phenotypes. Laws of grammar are universals underlying rules of grammar; they are part of the genotype. A law of grammar is an invariant over a class of particular rules of grammar.

As an example, consider the active/passive distinction in English:

Columbus discovered America.

America was discovered by Columbus.

Various formulations of the rule of passivization in English greatly differ in detail, but it is generally agreed that (1) passive in English applies to strings in which two noun phrases and a verb occur in this order: First Noun Phrase—Active Verb—Second Noun Phrase; and (2) passivization involves the postponing of the preverbal noun phrase and presposing of the postverbal noun phrase: Second Noun Phrase—Passive Verb—First Noun Phrase. In other words, the standard rule of passivization in English is stated in terms of the word order.

A rule of passivization such as this is clearly language specific. Since different languages have different word order—for example, in Malagasy, the direct object precedes the verb and the subject follows the verb—there must be distinct language-specific rules of passivization for these other languages. Furthermore, in languages such as Russian or Latin where word order does not have grammatical meaning, the rule of passivization is stated in terms of case endings.

From the perspective of the Transfer Principle, we face an empirical question: is there an invariant of the particular rules of passivization in particular languages? We believe that there is: a universal law that we call the *Law of Passivization*. This law is roughly characterized by two invariant processes: (1) a conversion of the predicate relation, and (2) a superposition of the second term of the converted predicate with an oblique term. This law will become clearer later, although its complete formulation is beyond the scope of this paper, and may be found in [18].

We claim that the Law of Passivization—and other laws as well—are *universal*. But in fact, in some cases a law is simply not applicable to certain languages, and the Law of Passivization is no exception: the active/passive correspondance does not exist in some languages. Our use of the term "universal" is from the perspective of treating the theory of grammar as a highly abstract branch of linguistics whose goal is to establish a single hypothetical grammatical system from which all possible structures and entities may be deduced. Universal laws hold for this abstract system, but concrete languages are in many cases subsets of the whole system, and thus certain laws may not be applicable.

This should not deter us from seeking universal laws, for two important reasons. First, the universal laws allow us to recognize common characteristics of otherwise very different languages. Second, they allow us to imagine, theoretically at least, what the instantiation of particular laws might look like in languages for which they are currently not applicable.

Principle 3 The Principle of Semiotic Relevance.

The only distinctions that are semiotically relevant are those that correlate with the distinctions between their signs, and, vice versa, the only distinctions between signs that are relevant are those that correlate with the distinctions between their meanings.

One can never overstate the significance of the Principle of Semiotic Relevance. If one wants to present de Saussure's doctrine in a single theoretical statement, the Principle of Semiotic Relevance is it. This principle defines the essence of linguistic reality, and is a keystone of the semiotic study of language.

The Principle of Semiotic Relevance is a powerful constraint on the theory of grammar, and not all linguistic theories conform well with this principle. For example, *generative phonology* [13, 3, 7] considers only the sound patterns of morphemes, completely disregarding their meanings. As a result, it wrongly identifies certain morphemes by posting fictitious relationships between them. The fundamental error of generative phonology is that is generates away cognate forms based entirely on formal criteria without regard to the meanings of the forms. For example, disregard of the meanings of the forms of morphemes leads to a confusion between synchrony and diachrony [18].

The opposite error is encountered in *generative semantics*, which fails to support distinctions in meanings with concomitant distinctions in phonic expressions. Consider, for instance, the famous McCawley's analysis of 'kill' as a causative verb in English. In a bracketed notation this analysis reads: *(cause (become (minus alive)))*; that is, "cause becomes minus alive," which is meant to be a semantic componential analysis of the verb 'kill'. This analysis is weak because it is based on the idea that given a possible paraphrase of the verb 'kill', it must therefore *ipso facto* be considered a causative verb. In accordance with

the Principle of Semiotic Relevance, any difference between linguistic meanings must be correlated with differences between phonic expressions. Real causative verbs are characterized by appropriate phonic markers as in the forms sit : set (I sit by the table, I set the table,) and fall : fell (the tree falls, the lumberjack fells the tree). The verb 'kill' has none of these phonological markers of the causative meaning.

Linguistic meaning is vital for communication, and is an essential aspect of every use of language; but the linguistic meaning does not constitute the total meaning of a sentence or word. Consider the sentence 'Garry Kasparov and I.B.M.'s computer Deep Blue came to draw in the fourth game yesterday.' The linguistic meaning of the sentence is determined by the dictionary and the rules of the grammar of English. But the sentence means more than that. A man who knows chess can infer from the context of the word 'game' that it was the game of chess. He may also infer that Kasparov and Deep Blue had played three games before the game yesterday. He may infer further that Deep Blue is a superstrong chess program because Kasparov is the world champion of chess. A man who does not know chess cannot infer from the meaning of the word 'game' that it was a chess game. From The sentence 'John killed a bear' we infer that John caused a bear not to be alive, but causation is an inferential meaning that is parasitic on the linguistic meaning of kill.

The total meaning of a sentence or word is a compound containing the linguistic meaning combined with other kinds of meaning just as a chemical compound contains a certain substance combined with other substances. To isolate a certain substance from other substances, one uses chemical reagents. The analysis of meaning is mental chemistry. The chemical reagent of the linguist is the Principle of Semiotic Relevance. Using it, the linguist isolates the linguistic meaning in its pure form.

Principle 4 The Unity of Syntactic and Semantic Representation.

Syntactic and Semantic representation cannot be separated from each other; they constitute a unique representation, called contensive representation.

This principle is a corollary of the Principle of Semiotic Relevance. It follows from this principle that any distinction in semantic representation must correlate with a distinction in syntactic representation, and vice versa: any distinction in syntactic representation must correlate with a distinction in semantic representation. A system of contensive representation is called *contensive syntax*.

Contensive syntax is a new concept which should not be confused with semantics. The existence of semantic rules presupposes the existence of syntactic rules. In contrast, contensive syntax is a unitary system of rules which is, so to speak, a chemical bond of structure and meaning. Just as water is a completely new substance in comparison with hydrogen and oxygen taken separately, so contensive syntax is a completely new entity in comparison with structure and meaning, which taken separately are not part of linguistics; they are part of logic.

The fundamental constraint on the combination of signs is the Sign Combination Principle.

Principle 5 The Sign Combination Principle.

A sign, called an operator, combines with one or more signs, called its operands, to form a new sign, called its resultant, on condition that its meaning is incomplete and needs to be supplemented by meanings of other signs.

For example, verbs and adjectives are operators with respect to nouns because meanings of verbs and adjectives are incomplete and are in need of supplementation by meanings of nouns. Consider 'boy' or 'paper.' The meanings of these nouns are complete. Take now 'writes' and 'white.' We ask: Who writes? What is white? The meanings of the words are incomplete. They need to be supplemented by meanings of nouns such as 'boy' or 'paper'. In 'the boy writes' the verb 'writes' is an operator and 'the boy' is its operand; in 'white paper' the adjective 'white' is an operator and 'paper' is its operand. Similarly, the meaning of prepositions is incomplete without supplementation by meaning of nouns; therefore prepositions are operators with respect to nouns. In 'on the table,' 'on' is an operator and 'the table,' its operand. Furthermore, the meaning of a conjunction is incomplete, and needs to be supplemented by the meaning of words belonging to basic word categories—nouns, adjectives, verbs, adverbs, or complete sentences. Therefore a conjunction is an operator with respect to expressions of all these categories: in 'black and white,' 'and' is an operator with respect to 'black' and 'white.'

In a later section we will see more elaborate examples of the Sign Combination Principle, including chains and hierarchies of meaning supplementations.

Principle 6 The Principle of Monotonic Constructions.

Any combination of linguistic units has a unique construction; in algebraic terms, any combination of linguistic units is non-associative.

Being a corollary of the Sign Combination Principle, the Principle of Monotonic Constructions is of paramount importance for linguistics. It excludes systems such as *generalized categorial grammar* [14], whose associativity means that a sentence can be bracketed in every possible way. Moorgat motivates the use of the associative calculus as follows:

The application analysis for 'John loves Mary' is strongly equivalent to the conventional phrase-structure representation for a sequence subject transitive verb—direct object, with the transitive verb and the direct object grouped into a VP constituent. Suppose now that we are not so much interested in constituent structure, as commonly understood, but rather in the notion of derivability, that is, in the question: Given a sequence of input types (viewed as sets of expressions), what type(s) can be derived from the concatenation of the input sentences? It will be clear that the result type S would also be derivable in the transitive verb had been assigned the type $NP \setminus (S/NP)$ instead of $(NP \setminus S)/NP$ [14, p. 148].

Associativity considerably simplifies the construction of mathematical models of language, but it distorts linguistic reality. Associativity is motivated primarily by convenience: an associative calculus is much more convenient for parsing a string of words in a purely mechanical fashion. The trouble is, as follows from the Sign Combination Principle, the sentences of a natural language have a nonassociative structure. Thus if we want to understand their structure, we have no choice but to construct a non-associate calculus.

Principle 7 The Principle of Type Assignment.

Every sign of the applicative system is assigned a type which defines its syntactic function.

The Principle of Type Assignment is subject to the following conditions:

- 1. Inclusion. Every atomic sign is assigned a characteristic type.
- 2. *Exclusion*. No sign belongs to more than one characteristic type.
- 3. *Superposition*. Every sign can be assigned a complementary type superposed on its characteristic type.
- 4. *Polymorphism.* Some signs can be assigned variable types. The range of a variable type includes concrete types having related functions.

Principle 8 The Superposition Principle.

If in a given context C a unit A takes on the function of the unit B as its complementary function, a syncretic unit $\langle A/B \rangle$ is formed. We say that A and B are superposed in the syncretic unit $\langle A/B \rangle$, and we call the operation of forming $\langle A/B \rangle$ the superposition of A with B. Given $\langle A/B \rangle$, A is called the basis, and B the overlay.

Superposed types are governed by the following principles:

- 1. Existence. The unit $\langle x/y \rangle$ exists in a given context C if the unit x is superposed with the unit y.
- 2. *Identity*. A superposed unit is distinct from its basis. Two superposed types are the same only if their bases and overlays are the same.
- 3. Inheritance. In any context C in which a superposed unit exists, it has those normal properties possessed by its basis.

Let us turn to examples of superposition. Consider the word 'lion'. The characteristic meaning of 'lion' is the name of an animal. But in combination with some words it takes on the meaning 'a famous and important person,' as in a 'literary lion'. The characteristic meaning of black is 'of the darkest color,' but the word may take on the meaning 'very bad,' as in 'black despair'. These are examples of the action of modifying contexts. A modifying context changes the meaning of the word with which it comes: the word becomes synonymous with some other word or expression. In our examples, the word 'lion' is synonymous with 'a famous and important person' in the context of literary; and the word 'black' is synonymous with 'very bad' in the context of despair. Due to the action of its modifying context the meaning of a word becomes figurative, representing a syncretism of two meanings: the initial meaning of the word and the meaning of the expression with which the word is synonymous. This is a case of polysemy. Due to the action of the modifying context the word becomes polysemous.

Nouns and adjectives seem to behave in a similar way: in some contexts they fulfill the role of the argument of a predicate, in other contexts, the role of an attribute of a noun. If we classify nouns and adjectives as polymorphic, then we must admit that their polymorphism is identical and that nouns and adjectives are identical at the level of their phrasal projection. But the resulting type ambiguity of lexical classes would then conflict with the generally accepted notion of lexical classes as morphologically and syntactically distinct entities. In search of a plausible explanation, we arrive at the hypothesis of a hierarchy of syntactic types assigned to each lexical class. It is this hierarchy that is explained by the Superposition Principle.

This analysis reveals the opposition between the noun and the adjective: the characteristic type of the noun is the complementary type of the adjective, and, conversely, the characteristic type of the adjective is the complementary type of the noun. A sign with a complementary type superposed on its characteristic type displays duality: it takes on the properties of the complementary type superposed on its characteristic type but retains at least part of properties of its characteristic type.

4 Parsing Natural Language Based on AUG

To understand the way in which parsing using 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 0xy, 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:

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

More generally, we can use the following rule to describe the application of one phrase, p of type 0xy, to another, q of type x:

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

Clearly, types of the form $\Box xy$ 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".



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:

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

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 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 described earlier.

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.

5 An NLP Prototype Written in Haskell

Our NLP prototype was written in Haskell [10], a standard for non-strict purely functional programming languages. Tutorial information on Haskell may be found elsewhere [1, 9]; in the following discussion we assume basic familiarity with the language. 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. As mentioned earlier, Curry himself was interested in the study of linguistics [4].

5.1 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.

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 a way to display typed tree structures, and so we define a function:

> drawTTree :: TTree -> String

to display a typed tree in the form shown earlier.

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 later sections:

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])]
```

5.2 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 $[w_1, \ldots, w_n]$. 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-1times 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.

5.3 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:

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.³ To allow for variations in word order, we consider both the possibility that 1 is applied to \mathbf{r} , and also that \mathbf{r} is applied to 1:

```
> 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,[0 x y]) (b,[x]), [y]) | (0 x y)<-ts, z<-ss, x==z ]</pre>
```

The expression (op (a, $[0 \times y]$) (b, [x]), [y]) here corresponds to the rule that, if a has type 0 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 5.1 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)
```

³ This limitation is not as severe as it might sound, linguistically, since *currying* permits application to several arguments. The parse described earlier involving the word 'in', with type **OTOOTSOTS**, is an example of this, as are transitive and ditransitive verbs, having types **OTOTS** and **OTOTOTS**, respectively.

We comment on these parses in more detail in Section 6.

For larger sentences, however, the definition of **ttrees** is not efficient enough. Fortunately, there is a much more efficient algorithm (described in [11]) based on *tabulation* [2]. With this change, our Haskell interpreter takes just a second to determine that there are 60 different parses of the 19 word sentence **long** given earlier, and this result would be at least an order of magnitude faster if it were compiled. (The original program took over 8 hours to achieve the same task!)

6 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 a deeper underlying 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 that cannot be distinguished by their grammatical structure alone, in which case some form of semantic analysis may be needed to resolve any ambiguities. ? explain "my old friend who comes from Moscow"

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



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



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



Fig. 1. Parsing the phrase 'my old friend who comes from Moscow'.

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.

6.1 A Refined Domain of Types

The datatype **Type** used to capture AUG types is actually a simplified version of that used in AUG. Thus we have extended it to the following:

data Type = T | S | T1 | T2 | T3 | O Type Type | Sup Type Type

The new constructor Sup is for superposition, and will be explained later. The new constructors T1, T2, and T3 capture a refined viewpoint of noun phrases called *primary*, *secondary*, and *tertiary* terms, respectively. This refinement is then reflected in the types of words and phrases that expect arguments or return results in this refined set. For example, the types of intransitive, transitive, and ditransitive verbs are given by 0 T1 S, 0 T2 (0 T1 S), and 0 T3 (0 T2 (0 T1 S)), respectively.

To see the effect of this change, under the original scheme the sentence 'he knew that his mother was ill' has two valid parses:





But the second of these is nonsense, since it reverses the roles of the noun phrases 'he' and 'that his mother was ill' with respect to the verb 'knew'. However, by declaring 'he' as T1, 'that' as OST2, and 'knew' as OT2OT1S, we arrive at the single sensible parse:



6.2 Superposition

An important aspect of AUG is its ability to deal with a word or phrase serving several simultaneous grammatical roles in a single sentence. The best example of this is the *gerund*, and we have implemented the rules to make this work. As an example, consider the sentence 'I see her coming home'. Here the word 'coming' serves both as an object (action) that is being seen, and as a verb applied to a secondary term 'home'. To represent phrases having more than one meaning, AUG uses the notation $\langle t_1/t_2 \rangle$ which is read: "the type t_1 superposed on type t_2 ." The typing rules for superposition are given in Figure 2.

We have implemented these rules, using the constructor Sup in the revised Type datatype to represent superposed types. As an example of the new parser in action, the parse of the sentence given above is:

$p:: 0 x \langle y/x \rangle q:: x$	$p:: \mathtt{O} x y q:: \langle x/z angle$	$p:: O xy q:: \langle z/x \rangle$
$pq::\langle y/x angle$	pq::y	$pq::\langle z/y angle$
$p::\langle 0 xy/z \rangle q::x$	$p::\langle z/\mathtt{O} xy angle q::x$	$p::\langle 0 x y/z \rangle q::\langle x/u \rangle$
pq::y	$pq::\langle z/y angle$	pq::y





6.3 Expanded Vocabulary

We have also expanded the vocabulary and dictionary search mechanism in several ways:

- 1. We have increased the number of words considerably; currently the vocabulary contains over 2000 words.
- 2. Many of these words have multiple types. For example, 'branch' has both types T and $\mathsf{OT1S}.$
- 3. We have implemented a simple form of singular/plural word inference.

6.4 Passivization

As a final extension of our basic scheme, we have added the notion of *passivization* to our parser. With this extension, we have defined a function **passive** that first parses a source string, passivizes the abstract parse, then regenerates the string in passive form. For example:

? passive "my friend will bring hats home" "hats will be brought home by my friend"

Note that this also required dealing with verb tenses, which we implemented at least to a limited extent. Note also the introduction of the word 'by' when moving to the passive form. Overall this is not a trivial transformation, and is a promising indication of the viability of our approach.

7 Conclusion

The history of computer science, mathematical logic, and linguistics presents a striking parallel with respect to their approach to language. Computer science constructs programming languages. Metamathematics is concerned with a critique of existing languages of mathematics and with constructing better languages. Linguistics is concerned with developing formal structures modeling natural languages.

Every language–a programming language, a language of logic or mathematics, or a natural language—depends on a physical substratum, which makes the existence of the language possible. The physical structure of programming languages is a reflection of the physical structure of the computer. The physical structure of natural language is so-called phrase structure which reflects the action of the human machine: the organs of speech. Thus natural language has a linear phrase-structure. The essential structure of natural language, its functional structure, is independent of the linear phrase-structure, because the raison d'etre of language, the sole reason for the existence of language, is to be a tool for the expression of thought and communication. Hence the need for a functional structure of language.

In computer science, mathematical logic, and linguistics we observe a consistent trend towards liberation from dependence on the physical structure of languages towards the representation of its pure functional structure. In computer science we observe three stages: (1) languages reflecting physical structure of computers; (2) imperative languages: a mixture of physical and functional structure; and (3) functional (applicative) languages: pure functional structure. In mathematical logic we observe two stages: (1) concatenative languages, tied to the linear representation of abstract systems; and (2) applicative languages, independent of the linear representation of abstract systems. Finally, in linguistics we observe two stages: (1) phrase-structure grammar, a mixed system tied to the linear representation/physical system; and (2) applicative grammar, a pure functional system. We also observe a strong correspondence between the functional structure of new functional programming languages such as Haskell and the functional structure of genotype, the underlying language of AUG.

8 Acknowledgements

We would like to thank Mark Jones, now at Nottingham University, for doing much of the initial Haskell programming on this project when he was at Yale, and who should probably be a co-author if time constraints had permitted it for him. Also thanks to the second author's funding agencies, DARPA under grant number F30602-96-2-0232, and NSF under grant number CCR-9633390.

References

- R. Bird and P. Wadler. Introduction to Functional Programming. Prentice Hall, New York, 1988.
- 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, pages 17–35. Prentice-Hall International Series in Computer Science, 1994.
- Noam Chomsky. On the notion 'rule of grammar'. In *Proceedings of Symposium* in Applied Mathematics, volume 12 (Structure of Language and Its Mathematical Aspects). 1961.
- H.B. Curry. Some logical aspects of grammatical structure. In Structure of language and its mathematical aspects. American Mathematical Society, Providence, 1961.
- H.B. Curry and R. Feys. Combinatory Logic, Vol. 1. North-Holland, Amsterdam, 1958.
- 6. R. Frost and J. Launchbury. Constructing natural language interpreters in a lazy functional language. *The Computer Journal*, 32(2):108–121, April 1989.
- 7. Morris Halle. The Sound Pattern of Russian. Mouton, The Hague, 1959.
- P. Hudak. Conception, evolution, and application of functional programming languages. ACM Computing Surveys, 21(3):359–411, 1989.
- P. Hudak and J. Fasel. A gentle introduction to Haskell. ACM SIGPLAN Notices, 27(5), May 1992.
- 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.
- M.P. Jones, P. Hudak, and S. Shaumyan. Using types to parse natural language. In Proceedings of Glasgow Functional Programming Workshop. IFIP, Springer Verlag, 1995.
- Stanislaw Lesniewski. Grundzuge eines neuen Systems der Grundlagen der Mathematik. Fundamenta Mathematicae, 14:1–81, 1929.
- Theodore M. Lightner. Generative phonology. In William Orr Dingwall, editor, Survey of Linguistic Science, pages 489–574. Linguistics Program, University of Maryland, 1971.
- M. Moortgat. The generalized categorial grammar. In Flip G. Droste and John E. Joseph, editors, *Linguistic Theory and Grammatical Description*, pages 489–574. John Benjamins Publishing, Amsterdam/Philadelphia, 1991.
- Richard Montague. Formal philosophy. In R.H. Thomason, editor, Selected writings of Richard Montague. Yale University Press, New Haven, CT, 1974.
- 16. Sebastian Shaumyan. Strukturnaja lingvistika, 1965.
- Sebastian Shaumyan. Applicative grammar as a semantic theory of natural language. University of Chicago Press, 1977.
- Sebastian Shaumyan. A Semiotic Theory of Language. Indiana University Press, 1987.
- 19. 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*. Dakota State University, Madison, Dakota, 1991.

This article was processed using the LAT_FX macro package with LLNCS style