A look at Grammatical Framework:
A type-theoretic approach to linguistics

Ricky Elrod
Youngstown State University
December 7, 2016

Type Theory

Introductory Type Theory

- Type theory is a branch of math, computer science, and logic.

Introductory Type Theory

- Type theory is a branch of math, computer science, and logic.
- Its applications reach far beyond those fields.

Introductory Type Theory

- Type theory is a branch of math, computer science, and logic.
- Its applications reach far beyond those fields.
- Created to eliminate mathematical paradoxes in certain branches of math.

Introductory Type Theory

- Type theory is a branch of math, computer science, and logic.
- Its applications reach far beyond those fields.
- Created to eliminate mathematical paradoxes in certain branches of math.
- Now used in computer science...
  - as a way to eliminate software bugs by proving software correct
  - as a way to formalize semantics of programming languages
  - in security-critical applications (financial contracts, cryptocurrency)

Introductory Type Theory

- In a type-theoretic system, a “term” (usually a mathematical object) is assigned a specific “type.”

Introductory Type Theory

- In a type-theoretic system, a “term” (usually a mathematical object) is assigned a specific “type.”
- Operations are restricted to terms of certain types.

Introductory Type Theory

- In a type-theoretic system, a “term” (usually a mathematical object) is assigned a specific “type.”
- Operations are restricted to terms of certain types.
- A typing judgement (written `t : T`) is a statement that the term `t` has type `T`.

Introductory Type Theory

- In a type-theoretic system, a “term” (usually a mathematical object) is assigned a specific “type.”
- Operations are restricted to terms of certain types.
- A typing judgement (written `t : T`) is a statement that the term `t` has type `T`.
- For example, if we call the type of natural numbers `nat`, then inhabitants of this type are `0, 1, 2, 3, ...`
  - We can say, for example, that `2 : nat` is well-typed.

Introductory Type Theory

functions

- Type theories have a notion of “functions.”

Introductory Type Theory

functions

- Type theories have a notion of “functions.”
- These are denoted with an arrow: $\to$

Introductory Type Theory

functions

- Type theories have a notion of “functions.”
- These are denoted with an arrow: $\rightarrow$
- For example, we can discuss a function that adds 2 to a given natural number and returns the result. This function would have type $\texttt{nat} \rightarrow \texttt{nat}$.
  - That is, the domain of the function is $\texttt{nat}$ and the value returned from it has type $\texttt{nat}$.

Grammatical Framework

Connecting Type Theory to Linguistics

Aarne Ranta's "Grammatical Framework" - 2003

- Ranta describes his framework as "a special-purpose functional [programming] language for defining grammars."

Connecting Type Theory to Linguistics

Aarne Ranta's "Grammatical Framework" - 2003

- Ranta describes his framework as "a special-purpose functional [programming] language for defining grammars."
- It makes use of a well-known type theory that is used in many proof-assistant programming languages today.

Two Kinds Of Syntax

- Abstract: describes a hierarchy for small components of the language to be glued together.

Two Kinds Of Syntax

- Abstract: describes a hierarchy for small components of the language to be glued together.
  - Easier for computers to process. Just traverse the hierarchy like any other tree structure.

Two Kinds Of Syntax

- Abstract: describes a hierarchy for small components of the language to be glued together.
  - Easier for computers to process. Just traverse the hierarchy like any other tree structure.
  - Harder to generate an abstract syntax for natural languages.

Two Kinds Of Syntax

- Abstract: describes a hierarchy for small components of the language to be glued together.
  - Easier for computers to process. Just traverse the hierarchy like any other tree structure.
  - Harder to generate an abstract syntax for natural languages.
- Concrete: describes what the end-user (programmer, speaker, writer, etc.) works with.

Grammatical Framework

- Grammar-based by default (symbolic approach)

Grammatical Framework

- Grammar-based by default (symbolic approach)
- Grammars can relate several languages at the same time.

Grammatical Framework

- Grammar-based by default (symbolic approach)
- Grammars can relate several languages at the same time.
- The system works like a programming language compiler:
  - A string is “parsed” into a tree structure that the compiler knows how to traverse (“abstract syntax tree” or AST).
  - The AST is used to generate a program in machine code that the computer can understand.

Grammatical Framework

- Grammar-based by default (symbolic approach)
- Grammars can relate several languages at the same time.
- The system works like a programming language compiler:
  - A string is “parsed” into a tree structure that the compiler knows how to traverse (“abstract syntax tree” or AST).
  - The AST is used to generate a program in machine code that the computer can understand.
- GF grammars are more powerful than a typical compiler: They can describe natural language (i.e., they do not have to be context-free), they are reversible, and they are multilingual.

Grammatical Framework

Several ways to write grammars

- Backus-Naur Form (BNF) can be used as a subset of the GF language for creating context-free grammars.

Grammatical Framework

Several ways to write grammars

- Backus-Naur Form (BNF) can be used as a subset of the GF language for creating context-free grammars.
- For more advanced work, the full GF language must be used.

A simple context-free grammar

using GF's implementation of BNF

```
Pred.       Comment ::= Item "is" Quality;
This.       Item    ::= "this" Kind;
That.       Item    ::= "that" Kind;
Mod.        Kind    ::= Quality Kind;
Wine.       Kind    ::= "wine";
Cheese.     Kind    ::= "cheese";
Fish.       Kind    ::= "fish";
Very.       Quality ::= "very" Quality;
Fresh.      Quality ::= "fresh";
Warm.       Quality ::= "warm";
Italian.    Quality ::= "Italian";
Expensive.  Quality ::= "expensive";
Delicious.  Quality ::= "delicious";
Boring.     Quality ::= "boring";
```

Using the grammar

- We can now ask GF to parse a string which makes use of the grammar.

```
demoAbs> import demo.cf

Languages: demo
0 msec


demoAbs> parse "this delicious cheese is
   expensive"
Pred (This (Mod Delicious Cheese)) Expensive

1 msec
```

Using the grammar

- The string has been parsed into an AST.

Using the grammar

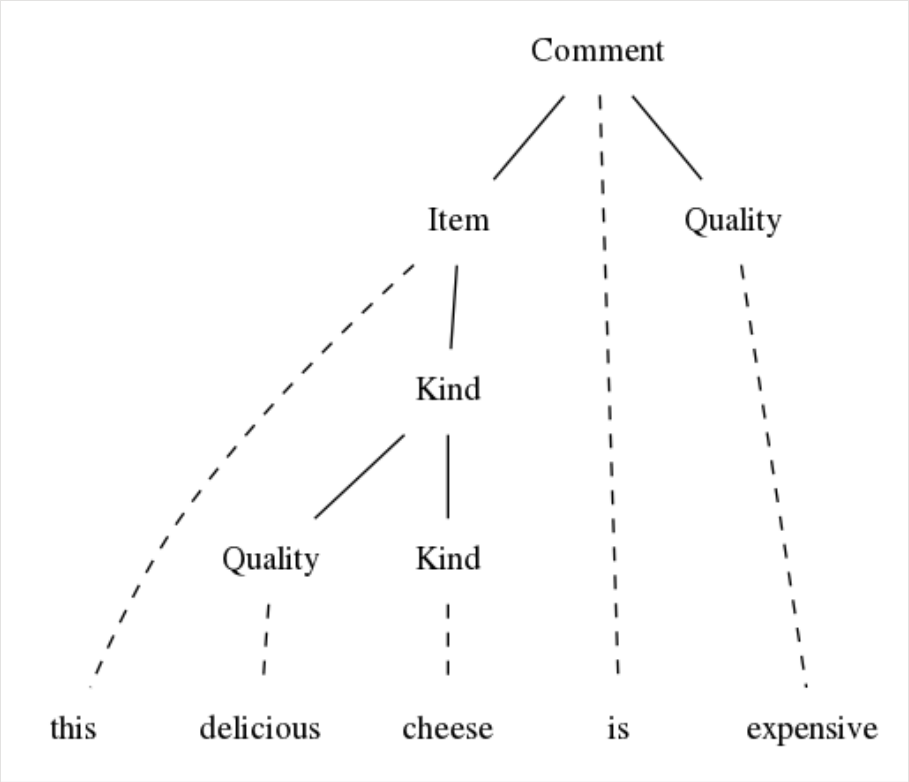
- The string has been parsed into an AST.
- The AST is unique: We can ask GF to “linearize” it back into its original string.

Using the grammar

- The string has been parsed into an AST.
- The AST is unique: We can ask GF to “linearize” it back into its original string.
- We can also ask GF to diagram the sentence (to the extent it can, with the information we have encoded thus far).

Sentence Diagramming

parse "this delicious cheese is expensive" | vp



The GF Language (without BNF subset)

- Abstractly define properties of the language.
- Provide concrete implementations of the abstract type.

Example from Ranta

Abstract `Food` grammar

```
abstract Food = {
    flags startcat = Comment ;
    cat
        Comment ; Item ; Kind ; Quality ;
    fun
        Pred : Item -> Quality -> Comment ;
        This, That : Kind -> Item ;
        Mod : Quality -> Kind -> Kind ;
        Wine, Cheese, Fish : Kind ;
        Very : Quality -> Quality ;
        Fresh, Warm, Italian,
            Expensive, Delicious, Boring :
                Quality ;
}
```

Example from Ranta

Concrete English implementation of `Food` grammar

```
concrete FoodEng of Food = {
    lincat
        Comment, Item, Kind, Quality = Str ;
    lin
        Pred item quality = item ++ "is" ++ quality ;
        This kind = "this" ++ kind ;
        That kind = "that" ++ kind ;
        Mod quality kind = quality ++ kind ;
        Wine = "wine" ;
        Cheese = "cheese" ;
        Fish = "fish" ;
        Very quality = "very" ++ quality ;
        Fresh = "fresh" ;
        Warm = "warm" ;
        Italian = "Italian" ;
        Expensive = "expensive" ;
        Delicious = "delicious" ;
        Boring = "boring" ;
}
```

Example from Ranta

Concrete Italian implementation of `Food` grammar

```
concrete FoodIta of Food = {
    lincat
        Comment, Item, Kind, Quality = Str ;
    lin
        Pred item quality = item ++ "é" ++ quality ;
        This kind = "questo" ++ kind ;
        That kind = "quel" ++ kind ;
        Mod quality kind = kind ++ quality ;
        Wine = "vino" ;
        Cheese = "formaggio" ;
        Fish = "pesce" ;
        Very quality = "molto" ++ quality ;
        Fresh = "fresco" ;
        Warm = "caldo" ;
        Italian = "italiano" ;
        Expensive = "caro" ;
        Delicious = "delizioso" ;
        Boring = "noioso" ;
}
```

Example from Ranta

Making use of it

```
> import FoodEng.gf FoodIta.gf
linking ... OK

Languages: FoodEng FoodIta
5 msec
Food> parse -lang=Eng "this delicious wine is Italian" | linearize -lang=Ita
questo vino delizioso é italiano

Food> generate_random | linearize -treebank
Food: Pred (That (Mod (Very Fresh) Cheese)) Delicious
FoodEng: that very fresh cheese is delicious
FoodIta: quel formaggio molto fresco é delizioso

Food> translation_quiz -from=FoodIta -to=FoodEng
Welcome to GF Translation Quiz.
The quiz is over when you have done at least 10 examples
with at least 75 % success.

quel pesce é molto molto caldo
>>> that fish is very very warm
> Yes.
Score 1/1
quel formaggio é caro
>>> that cheese is fresh
> No, not that cheese is fresh, but
that cheese is expensive

Score 1/2
```

Word Alignment

parse "this very warm cheese is Italian" | align_words