Rapid Development of Dialogue Systems by Grammar Compilation*

Björn Bringert
Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden
bringert@cs.chalmers.se

Abstract

We propose a method for rapid development of dialogue systems where a Grammatical Framework (GF) grammar is compiled into a complete VoiceXML application. This makes dialogue systems easy to develop, maintain, localize, and port to other platforms, and can improve the linguistic quality of generated system output. We have developed compilers which produce VoiceXML dialogue managers and ECMAScript linearization code from GF grammars. Along with the existing GF speech recognition grammar compiler, this makes it possible to produce a complete mixed-initiative information-seeking dialogue system from a single GF grammar.

1 Introduction

In current industrial practice, dialogue systems are often constructed using VoiceXML for dialogue management, context-free speech recognition grammars for input, with semantic tags for interpretation, and concatenation of canned text and output data for system responses. Developing several components which all need to cover the same concepts increases development costs. Having multiple interdependent components in formalisms with few automatic correctness and consistency checks also complicates maintenance, since any change in the coverage of one component may require changes in the others. Since all the components are language-specific, much effort is needed to port the system to a new language, and to keep the implementations for different languages in sync. The lack of a powerful method for output realization makes it hard to generate high-quality output, especially for languages with a more complex morphology than English.

We address these problems by specifying the system in a single high-level formalism, which is then compiled into the existing lower-level formalisms. The developer writes a GF abstract syntax module which defines the user input and system output semantics, and a concrete syntax module which describes how each construct in the semantics is represented in natural language. The GF grammar is then compiled to a complete VoiceXML application. The dialogue flow is determined by the abstract syntax (ontology) of the grammar. This is based on the idea by Ranta and Cooper (2004) that a proof editor for constructive type theory can be used to implement the information gathering phase of information-seeking dialogue systems.

In contrast to earlier rapid dialogue system development approaches, such as CSLU's RAD (McTear, 1999) and the GEMINI AGP (Hamerich et al., 2004), we use a compiler-like model instead of a graphical design environment. In addition, our development model is focused on the specification and realization of the inputs and outputs of the system, rather than on the dialogue flow or the underlying database. Compared to existing dialogue systems built with GF (Ericsson et al., 2006), our approach does not require an external dialogue manager.

*This work has been partly funded by the EU TALK project, IST-507802, and Library-Based Grammar Engineering, Swedish Research Council project dnr 2005-4211.

Proceedings of the 8th SIGdial Workshop on Discourse and Dialogue, pages 223–226,
Antwerp, September 2007.

2 Grammatical Framework

Grammatical Framework (GF) (Ranta, 2004) is a grammar formalism based on constructive type theory. GF separates grammar into *abstract syntax* and *concrete syntax*.

2.1 Abstract Syntax

The abstract syntax defines the ontology of the application, that is, *what* can be said. An abstract syntax contains category (**cat**) and function (**fun**) definitions. This is an example of a small abstract syntax:

cat Order; Size;
fun *pizza* : Size → Order; *small* : Size;

This allows us to construct an abstract syntax term *pizza small* of type Order. In addition to functions, abstract syntax terms can also contain *metavariables*, written ?. For example, the term *pizza* ? contains a metavariable of type Size.

2.2 Concrete Syntax

A concrete syntax defines *how* each abstract syntax construct is realized in a particular language. From a concrete syntax, the GF system can derive both parsing and realization components. A concrete syntax contains linearization type (**lincat**) and linearization (**lin**) definitions. The linearization type of a category is the type of the concrete syntax terms produced for abstract syntax terms in the given category. A linearization definition is a function from the linearizations of the arguments of an abstract syntax term to a concrete syntax term. Terms in concrete syntax can be records, strings, tables, and parameters. This is an example of a concrete syntax for the abstract syntax above:

lincat Order, Size = $\{s : \mathsf{Str}\}$;
lin *pizza* $x = \{s =$ "a" ++ $x.s$ ++ "pizza"$\}$;
small $= \{s =$ "small"$\}$;

3 An Example Dialogue System

This section shows a GF grammar from which a complete dialogue system (excluding the domain resources) can be derived automatically. For reasons of brevity, this system is very small. An extended version of this system is available online[1].

[1] http://www.cs.chalmers.se/~bringert/xv/pizza/

3.1 Abstract Syntax

The abstract syntax in Figure 1 describes the possible things that the user can say, in a semantic form. There is one category for each kind of input. In this application, the main input object is an Order. A order can in this small example only be for a number of pizzas, all of the same size and with the same topping. A number is "one" or "two", the sizes are "small" and "large", and the toppings are "ham" and "cheese". An example abstract syntax term in the Order category is: *pizza two small cheese*.

abstract Pizza = {
flags startcat = Order;
cat Order; Number; Size; Topping;
fun *pizza* : Number → Size → Topping → Order;
one, *two* : Number;
small, *large* : Size;
cheese, *ham* : Topping;
}

Figure 1: Abstract syntax for the example system.

3.2 Concrete Syntax

The concrete syntax in Figure 2 defines how the terms in the abstract syntax are realized (and inversely, how concrete syntax terms can be interpreted as representations of abstract syntax terms). For example, the linearization type of Topping is $\{s : \mathsf{Str}\}$, that is, a record with a single field s which contains a string. The linearization for *cheese* is the concrete syntax term $\{s =$ "cheese"$\}$.

The linearization type of Number contains a field n, which is used for agreement. The type of n is Num, defined by a **param** definition to be either Sg or Pl. In the linearization of *pizza*, the n field of the Number is used to inflect the noun "pizza".

An important feature of this grammar is that it allows partially specified input. While the utterance "two small pizzas with cheese" results in the abstract syntax term *pizza two small cheese*, the partial versions "two pizzas with cheese" (*pizza two ? cheese*), "two small pizzas" (*pizza two small ?*), and "two pizzas" (*pizza two ? ?*) are also allowed. The intention is that the system will ask follow-up questions to replace all metavariables with complete terms. This process is type-directed: the system asks for a subterm of the appropriate type. Partial input, imple-

```
concrete PizzaEng of Pizza = {
lincat Number = {s : Str; n : Num};
       Order, Size, Topping = {s : Str};
param Num = Sg | Pl;
printname cat
  Order   = "What would you like to order?";
  Size    = "What size pizzas do you want?";
  Topping = "What topping do you want?";
lin pizza n s ts = {s =
      n.s ++ variants {s.s; []} ++ pizza_N.s ! n.n
      ++ variants {"with" ++ ts.s; []}};
    one     = {s = "one"; n = Sg};
    two     = {s = "two"; n = Pl};
    small   = {s = "small"};
    large   = {s = "large"};
    cheese  = {s = "cheese"};
    ham     = {s = "ham"};
oper pizza_N = {s = table {Sg => "pizza";
                           Pl => "pizzas"}};
}
```

Figure 2: Concrete syntax for the example system.

mented with suppression, is thus used to achieve a mixed-initiative dialogue.

The **printname** definitions are used as prompts for each category.

3.3 Example Dialogues

The system generated from the grammar in the previous section allows dialogues such as the examples below. After each system action we show the information state, i.e. the current state of the abstract syntax term that we are constructing.

S: What would you like to order?
U: two pizzas
pizza two ? ?
S: What size pizzas do you want?
U: small
pizza two small?
S: What topping do you want?
U: ham
pizza two small ham

Here, more information is given in the first answer:

S: What would you like to order?
U: two pizzas with ham
pizza two ?*ham*
S: What size pizzas do you want?
U: small
pizza two small ham

3.4 Extending the Example System

Recursive structures One possible extension to the example system is to use a recursive structure to allow more complex orders:

```
cat Order; Item; [Item];
fun order : [Item] → Order;
    pizza : Number → Size → Topping → Item;
printname cat [Item] = "Anything else?";
lin order is = {s = is.s};
    ConsItem x xs =
      {s = x.s ++ variants {"and" ++ xs.s; []}};
    BaseItem = {s = "nothing" ++ "else"};
```

While this can be done with subdialogues and scripting in VoiceXML (by essentially writing by hand the code that we generate), it appears to be beyond the scope of standard practice. If we also add drinks as a kind of Item, the system will support dialogues such as this one:

S: What would you like to order?
U: one large pizza
order [*pizza one large*?, ?]
S: What topping would you like?
U: cheese
order [*pizza one large cheese*, ?]
S: Anything else?
U: one beer
order [*pizza one large cheese*, *drink one beer*, ?]
S: Anything else?
U: nothing else
order [*pizza one large cheese*, *drink one beer*]

System output At the end of the dialogue, we would like the system to give a response based on the output of some domain resource. For example, the pizza ordering system might return the price of the order. This could be used to construct a confirmation using an addition to the grammar:

```
cat Output;
fun confirm : Order → Number → Output;
lin confirm o p =
  {s = o.s ++ "costs" ++ p.s ++ "euros"}
```

Multilinguality To port a dialogue system to a new language, all that needs to be done is to write a new concrete syntax. For many languages, writing speech recognition grammars and realization functions is more complicated than for English. For example, Swedish adjectives agree with the gender and number of the noun they modify. GF's expressive concrete syntax makes it possible to implement such features with little effort, and if the GF Resource Grammar Library is used, it is as easy to write the Swedish grammar as the English.

Multimodality GF can be used to write multimodal grammars (Bringert et al., 2005). The extended online version of the example system uses a concrete syntax which linearizes pizza and drink orders to vector drawings to display graphical representations of the completed orders.

4 Implementation

The concrete syntax is compiled (Bringert, 2007) to an SRGS speech recognition grammar, with SISR semantic interpretation tags. This grammar has one category for each GF category.

The abstract syntax and the prompts from the concrete syntax are compiled to a VoiceXML application with one form for each GF category. Each such form takes an argument, which the caller sets to the currently known abstract syntax term. If the given term is a metavariable, input is requested in the appropriate speech recognition grammar category. For each subterm of the abstract syntax term returned by the semantic interpretation, a subdialogue call is made to the corresponding VoiceXML form.

The concrete syntax is also compiled to an ECMAScript program which can be used to linearize system outputs.

5 Future Work

Currently, the dialogue model is quite limited. For real-world use, more flexible dialogue management would be needed. The Trindi tick list (Bohlin et al., 1999) could be used to guide such work. Other possibilities could include support for dependently typed abstract syntax (Ranta and Cooper, 2004), a help system with automatically generated examples for each category, and context-sensitive prompt generation.

6 Conclusions

We have shown that GF grammars can be used to implement mixed-initiative information-seeking dialogue systems. From the declarative and linguistically powerful specification that a GF grammar is, we generate the interconnected components needed to run dialogue systems using industry standard infrastructure. Hopefully, this method can reduce the development and maintenance costs for dialogue systems, and at the same time improve their linguistic quality. The methods described in this paper are implemented as part of the open source GF system[2].

References

Peter Bohlin, Johan Bos, Staffan Larsson, Ian Lewin, Colin Matheson, and David Milward. 1999. Survey of Existing Interactive Systems. D 1.3, TRINDI.

Björn Bringert, Robin Cooper, Peter Ljunglöf, and Aarne Ranta. 2005. Multimodal Dialogue System Grammars. In *Proceedings of DIALOR'05, Ninth Workshop on the Semantics and Pragmatics of Dialogue*.

Björn Bringert. 2007. Speech Recognition Grammar Compilation in Grammatical Framework. In *Proceedings of the Workshop on Grammar-Based Approaches to Spoken Language Processing*.

Stina Ericsson, Gabriel Amores, Björn Bringert, Håkan Burden, Ann-Charlotte Forslund, David Hjelm, Rebecca Jonson, Staffan Larsson, Peter Ljunglöf, Pilar Manchón, David Milward, Guillermo Pérez, and Mikael Sandin. 2006. Software illustrating a unified approach to multimodality and multilinguality in the in-home domain. D 1.6, TALK.

Stefan Hamerich, Volker Schubert, Volker Schless, Ricardo de Córdoba, José M. Pardo, Luis F. d'Haro, Basilis Kladis, Otilia Kocsis, and Stefan Igel. 2004. Semi-Automatic Generation of Dialogue Applications in the GEMINI Project. In *Proceedings of the 5th SIGdial Workshop on Discourse and Dialogue*.

Michael F. McTear. 1999. Software to support research and development of spoken dialogue systems. In *Proceedings of Eurospeech'99*.

Aarne Ranta and Robin Cooper. 2004. Dialogue Systems as Proof Editors. *Journal of Logic, Language and Information*, 13(2):225–240.

Aarne Ranta. 2004. Grammatical Framework: A Type-Theoretical Grammar Formalism. *Journal of Functional Programming*, 14(2):145–189.

[2]See `http://www.cs.chalmers.se/~aarne/GF/`