A Grammar Formalism for Specifying ISU-based Dialogue Systems

Peter Ljunglöf and Staffan Larsson

Gothenburg University, Dept. of Linguistics,
Renströmsgatan 6, S-41255 Göteborg, Sweden
{peb,sl}@ling.gu.se

Abstract. We describe how to give a full specification of an ISU-based dialogue system as a grammar. For this we use Grammatical Framework (GF), which separates grammars into abstract and concrete syntax. All components necessary for a working GoDiS dialogue system are specified in the abstract syntax, while the linguistic details are defined in the concrete syntax. Since GF is a multilingual grammar formalism, it is straightforward to extend the dialogue system to several languages. Furthermore, the GF Resource Grammar Library can be used to write a single concrete instance covering 13 different languages.

1 Introduction

1.1 The Information-State Update Approach

The GoDiS dialogue manager [1] is based on formal semantic and pragmatic theories of dialogue, and provides general and fairly sophisticated accounts of several common dialogue phenomena such as interactive grounding, accommodation, multiple conversational threads, and mixed initiative. GoDiS is based on the Information State Update (ISU) approach to dialogue management [2]. The ISU approach, which has been developed over the last 10 years in several EU-funded projects, provides a generalization over previous theories of dialogue management and allows exploring a middle ground between sophisticated but brittle research systems, and robust but simplistic commercial systems. In the ISU approach, a dialogue manager is formalized as:

1. an information state (IS) type declaration, indicating which kinds of information the system needs to keep track of
2. a set of dialogue moves, i.e., abstract semantic representations of utterances
3. information state update rules, which:
   (a) update the IS based on observed dialogue moves
   (b) select moves to be performed by the system based on the current IS

The IS is a structured object which can contain e.g. sets, stacks, records, queues, etc. This allows efficient scripting of ISU system applications. In GoDiS, which is based on a theory of Issue-Based Dialogue Management (IBDM), a single

script (called a *dialogue plan*) can be used flexibly by the dialogue manager to allow for a wide range of dialogues. While it would in principle be possible to implement many GoDiS dialogues using finite state automata, the processes of implementing, debugging and modifying these systems would be prohibitive.

The main benefit of the IBDM account as implemented in GoDiS is the combination of advanced dialogue management and rapid prototyping (quick implementation, debugging and modification) enabled by cleanly separating general dialogue solutions from application-specific domain knowledge.

1.2 Examples of Flexible Dialogue.

Below are some examples of dialogues enabled by a single dialogue plan in the IBDM framework. The examples are taken from an implementation of a cellphone application. The dialogue plan, in this case the plan for changing a phone number in the phone book, specifies a default order of system actions to take:

1. ask for the name
2. ask for the type of number (home, cellphone, or work)
3. ask for the new number
4. change the phone book entry for the given name and type to the new number

Example 1. In the first example, after requesting the system to change a number, the user gives initiative to the system and just answers system questions:

U: Change phone number.
S: Okay, let's see. What name?
U: Lisa
S: Okay. Home number and work number are available. Which number do you want to change?
U: Home number
S: Okay. What is the new number?
U: 11 12 13.
S: Okay. I have now changed Lisa's home number to 11 12 13.

Example 2. Alternatively, the user can combine the request with answers to some as-yet-unasked questions. The default ordering of system actions in the dialogue plan can thus be overridden:

U: I would like to change Lisa's number to 11 12 13.
S: Okay, let's see. Home number and work number are available. Which number do you want to change?
U: The home number please.
S: Okay. I have now changed Lisa's home number to 11 12 13.

Example 3. It is also possible to just throw the system some information, and it will try to figure out which action the user wants the system to perform. If there is more than one candidate, the system will generate a clarification question. (Here we are also using a plan for making calls and a plan for searching for phone numbers):

U: Lisa's home number
S: Lisa's home number. I don't quite understand. Do you want to make a call, change a phone number or search for a phone number?
U: Change phone number
S: Okay. What is the new number?
U: 11 12 13.
S: Okay. I have now changed Lisa's home number to 11 12 13.

Example 4. If we provide the application with additional plans, e.g. for reading SMS messages, it is possible for the user to take new initiatives (ask questions or request actions) at any point in a dialogue. The system will enter into a sub-dialogue, and when the sub-dialogue is finished it will return to the previous dialogue topic. There is no intrinsic limit to the number of simultaneous conversational topics that the system can keep track of:

U: I would like to change Lisa's number to 11 12 13.
S: Okay, let's see. Home number and work number are available. Which number do you want to change?
U: Check my messages.
S: You have got this SMS: "Hi! I have a new home number, it is 11 12 13. Love, Lisa". Let's go back to changing an entry. Which phone number do you want to change?
U: The home number please.
S: Okay. I have now changed Lisa's home number to 11 12 13.

In addition to the above, general techniques for dealing with problems of perception, understanding and acceptance enable both user and system to provide feedback at any point in the dialogue. Such feedback can be either positive (signaling successful perception, understanding and acceptance) or negative (signaling problems).

We want to stress again that all the dialogues above use the same single dialogue plan. That is, the addition of that single plan to the application domain knowledge module enables many different dialogues.

1.3 Adding Natural Language to GoDiS

As previously mentioned, GoDiS enables rapid prototyping of systems with advanced dialogue behavior. However, the GoDiS dialogue manager only communicates with the outside world using semantic representations called *dialogue moves*. The designer of the dialogue system must implement a translation between natural language utterances and dialogue moves, be it through a simple lookup table, or an advanced feature-based grammar. If the system is speech-based, it also needs a statistical language model or a speech recognition grammar.

These components have to be maintained. If we add a new concept, e.g., a new dialogue plan, we have to add new grammar rules for handling the actions, questions, answers, etc., that are associated with the concept. Each entity in the database has to exist, both in the speech recognition component, in the grammar

and in the dialogue system. If the dialogue system is multilingual, we have to ensure consistency for each language.

There have been attempts of solving parts of these consistency problems. The Regulus grammar compiler [3] or the Grammatical Framework [4] can automatically create speech recognition grammars from a higher-level grammar, thus ensuring consistency between speech recognition and parsing. Both these formalisms have been used for building grammars for GoDiS systems [5].

One problem is still not sufficiently addressed: consistency between the dialogue system and the grammar. The dialogue moves that the grammar outputs from parsing have to conform to the dialogue moves that the GoDiS system recognizes; and the other way around: The grammar has to be able to translate dialogue moves from GoDiS into natural language utterances.

What we want is a single formalism where we can specify the complete dialogue system. There have already been some attempts of this, but not for ISU-based dialogue systems. In [6] it is shown that a simple GF grammar can be converted into a VoiceXML dialogue system. However, their translation can currently only handle small domains, and the resulting system has very limited dialogue handling capabilities. In this paper we show how a GoDiS dialogue system can be specified as a GF grammar. All components necessary for a full-fledged ISU-based dialogue system are then automatically generated from the grammar.

1.4 Grammatical Framework

Grammatical Framework [4] is a grammar formalism based on type theory. The main feature is the separation of abstract and concrete syntax, which makes it very suitable for writing multilingual grammars. A rich module system also facilitates grammar writing as an engineering task, by reusing common grammars.

The main idea of GF is the separation of abstract and concrete syntax. The abstract part of a grammar defines a set of abstract syntactic structures, called abstract terms or trees; and the concrete part defines a relation between abstract structures and concrete structures. This separation of abstract and concrete syntax is crucial for the treatment of dialogue systems in this article.

The abstract theory of GF is a version of Martin-Löf's [7] dependent type theory. A grammar consists of declarations of categories and functions. Categories can depend on other categories – the following declarations state that Request and Utterance are categories that depend on a Domain:

cat Domain
cat Request(Domain)
cat Utterance(Domain)

Functions are declared by giving argument and result types. Function declarations can also bind variables to be used in dependent types. Here we state that an Utterance can consist of a Request, provided that the share the same Domain:

fun request : (d:Domain)→ Request(d)→ Utterance(d)

Concrete Syntax. GF has a *linearization* perspective to grammar writing, where the relation between abstract and concrete is viewed as a mapping from abstract to concrete structures, called linearization terms.

Linearizations are written as terms in a typed functional programming language, which is limited to ensure decidability in generation and in parsing. The language has records and inflection tables; and the basic types are strings and inflection parameters. There are also local definitions, lambda-abstractions and global macro definitions. The parameters are declared in the grammar; they can be hierarchical but not recursive, to ensure finiteness.

The following things are declared in the concrete syntax:

- The *inflection parameters* have to be declared. E.g., a verb phrase request in a simple variant of Swedish can be in imperative or infinitive:
  param VerbForm = Imperative | Infinitive
- Each category should have a matching *linearization type*. E.g., a Swedish verb phrase request depends on the VerbForm:
  lincat Request = VerbForm ⇒ Str
  lincat Utterance = Str
- For each function in the abstract we define its *linearization function*. An utterance for our Swedish requests can either be a direct Imperative, or an indirect ("I would like to" followed by an Infinitive):
  lin request(req) =
  variants{req ! Imperative ; "jag vill" ++ req ! Infinitive ++ "tack"}
- A category can have an optional *default linearization*, which is used for unknown terms of that category:
  lindef Request =
  table{Imperative → "gör någonting" ; Infinitive → "göra någonting"}

With these example definitions, the possible linearizations of the incomplete term request(_) are "gör någonting" ("do something") and "jag vill göra någonting tack" ("I want to do something please").

Multilinguality and Resource Grammars. It is possible to define different concrete syntaxes for one particular abstract syntax. Multilingual grammars can be used as a model for interlingua translation, but also to simplify localization of language technology applications such as dialogue systems.

The abstract syntax of one grammar can be used as a concrete syntax of another grammar. This makes it possible to implement grammar resources to be used in several different application domains.

These points are currently exploited in the GF Resource Grammar Library [8], which is a multilingual GF grammar with a common abstract syntax for 13 languages, including Arabic, Finnish and Russian. The grammatical coverage is similar to the Core Language Engine [9]. The main purpose of the Grammar Library is as a resource for writing domain-specific grammars.

Note that for ease of presentation we do not make use of resource grammars in our running example. The interested reader is referred to [10], for a survey of the GF module system and resource grammars.

2 The GoDiS Dialogue Manager

In this section we give a short description of the building blocks of the GoDiS dialogue manager. The purpose of this description is to give details on how to specify a GoDiS system. We are not trying to explain the internals of the dialogue manager, which is described thoroughly in [1].

The GoDiS system communicates with the user via *dialogue moves*. There are three main dialogue moves – requesting actions, asking questions and giving answers. All three moves take one argument – the action, question or answer that the move is requesting, asking or giving.

Apart from the three main moves there are also different kinds of feedback moves – confirmations, failure reports and interactive communications management. We will not dwell into how these moves function, except for noting that they are important for the dialogue flexibility demonstrated in section 1.2.

The basic building blocks in GoDiS are individuals, sorts, one-place predicates and actions:

- The sorts are ordered in an hierarchy of sub- and supersorts. Each predicate has a domain which is a specific sort.
- Each individual e belongs to a specific sort s, written e : s.
- A predicate p (with domain s') can be applied to an individual e:s, where s is a subsort of s', to form a proposition p(e). A proposition can be used in an answer, answer(p(e)), or a y/n-question, ask(?p(e)).
- A collection of y/n-questions can be asked as an alternative question, ask({?p(e), ?p(f), ...}).
- A predicate p can be eta-expanded to a wh-question ?x.p(x). Wh-questions can be asked, ask(?x.p(x)).
- An action a can be requested, request(a). After the action has been performed it is confirmed, confirm(a), or a failure is reported, report(fail(a,...)).
- From an action a or a question q we can form the special propositions action(a) and issue(q).[1] These propositions are mainly used when asking the user what to do, or in feedback moves.

To specify a GoDiS dialogue system, we have to give the following information:

- The sortal hierarchy, i.e., the subsort relation.
- The individuals and the sorts they belong to.
- The predicates and their domains.
- The actions.
- The dialogue plans.

Apart from these things we have to have an interface for communicating with the device. The only thing we assume about this interface is that it can accept actions (using the dev_do plan construct) and queries (using dev_query).

[1] The propositions can be read approximately as "action a should be performed" and "question q should be resolved", respectively.

Dialogue Plans. Dialogue plans have already been touched upon in section 1.2. They convey what the system can do and/or give information about. A dialogue plan is a receipt for the system, so it knows how to answer a specific question, or how to perform a given action. The dialogue plans can roughly be divided into three different kinds – actions, issues and menus.

An *action plan* is when the user wants to perform an action, e.g., change the number of a contact in the phone book. Action plans are usually built in the same way. First the system asks some questions to get enough information, and then the action is performed. As an example, this is a more formal version of the plan in section 1.2:

changeNumber: findout(?x.nameToChange(x))
findout(?y.typeToChange(y))
findout(?z.newNumber(z))
dev_do(changeNumber)

After the plan has finished, GoDiS reports to the user about the success or failure of the action.

An *issue plan* is when the user has (explicitly or implicitly) asked a question, which the system should answer. Issue plans usually follow the same pattern as action plans, except that instead of telling the device to execute an action, it is given a query to solve. Here is the example plan for searching for phone numbers:

?x.searchForNumber(x): findout(?y.nameToSearch(y))
findout(?z.typeToSearch(z))
dev_query(?x.searchForNumber(x))

The result of the query is an answer to the question, which GoDiS automatically reports to the user.

A special kind of action plan is the *menu*, where the user can select from any of a given number of sub-plans which the system then performs. Note that these sub-plans can be menus themselves, which gives a hierarchy of menus.

managePhonebook: findout({ ?action(addContact)
?action(deleteContact)
?action(changeNumber)
?issue(?x.searchForNumber(x))
?issue(?y.searchForName(y)) })

3 Specifying a GoDiS System as GF Abstract Syntax

In this section we show how all necessary components of a GoDiS dialogue system can be specified in the abstract syntax of a GF grammar. All GoDiS components can be automatically extracted from the grammar.

3.1 Menus, Actions and Issues

In our GF grammar we define a category Menu, and three categories depending on Menu, reflecting the actions, issues and sub-menus in a plan.

```
cat Menu
cat Action(x)          [x : Menu]
cat Issue(x)           [x : Menu]
cat SubMenu(x,y)    [x,y : Menu]
```

Each action and issue in our dialogue specification belongs to a menu. Now, the first thing we have to do is to define the menus in our dialogue system:

```
fun mainMenu, makeCall, managePhonebook : Menu
```

An action plan is specified by giving a function with result category Action(m) where m : Menu. An example is the plan for changing the phone number:

```
fun changeNumber : nameToChange → typeToChange → newNumber →
                                                  Action(managePhonebook)
```

An issue in GoDiS is a wh-question ?x.P(x). This is reflected in the GF grammar where all issues are functions with the result Issue(m). Here is the issue plan for searching for a contact's phone number:

```
fun searchForNumber : nameToSearch → typeToSearch →
                                       number → Issue(managePhonebook)
```

Note that there is a crucial difference between the arguments. All arguments except the last one represent information which the system asks the user for. The last argument represents the final answer of the query.

Each menu in the specification corresponds to a menu plan in GoDiS. The elements of a menu are specified by the argument m to the dependent types Action(m) and Issue(m). E.g., the menu managePhonebook consists of five choices, of which changeNumber and searchForNumber are already specified above. With this solution we do not have to specify the menu plans directly, but they can be deduced automatically from the menu argument to each action and issue.

Finally, the mainMenu in our example asks whether we want to make a phone call, or manage the phone book. Both these alternatives are menus themselves. This is specified by creating instances of the SubMenu type:

```
fun makeCallSubMenu : SubMenu(mainMenu,makeCall)
fun managePhonebookSubMenu : SubMenu(mainMenu,managePhonebook)
```

3.2 The Dialogue System Ontology

Everything else in the GF grammar specifies the ontology of the dialogue system. From the grammar we can extract the sorts and the sortal hierarchy, the individuals and the sorts they belong to, and the predicates and their domains.

Sorts. In our simple example we want to have two GoDiS sorts, names and phone numbers. Names are defined as a simple database:

fun anna, bert, charles, diane : name

In our setting a phone number is simply a sequence of small numbers (i.e., numbers below 100):

fun single : smallNumber → number
fun cons : smallNumber → number → number
fun 0, 1, 2, . . . , 99 : smallNumber

Each of the GF types is automatically translated to a GoDiS sort, and each instance becomes a GoDiS individual. The complex functions create non-atomic individuals, so these are the accepted numbers in our GoDiS application:

single(n) : number **if** n : smallNumber
cons(n,m) : number **if** n : smallNumber **and** m : number

Note that the sort smallNumber will be created, which we do not use at all in our application. But this is no problem since it doesn't interfere with the sorts we are using.

User Answers. Not all sorts are intended to be used in communication. E.g., we do not want the user to give answers of the form answer(smallNumber(. . .)), but only of the form answer(number(. . .)). Therefore the grammar writer has to specify which sorts can be uttered as answers, by supplying the category Answer:

fun answerName : name → Answer
fun answerNumber : number → Answer

With these two definitions, the user can give answers containing names and phone numbers, but not small numbers.

Coercions and Subsorts. Each type that occurs as an argument in an Action or an Issue reflects a system-initiated question. E.g., the action for calling a phone number is:

fun callNumber : numberToCall → Action(makeCall)

From this specification, numberToCall will be translated to a one-place predicate in GoDiS. But GoDiS also needs to know the domain of this predicate. This is specified by a *coercion* function in GF:

fun coerceNumber : number → numberToCall

A function is a coercion if it, *i*) takes exactly one argument, and *ii*) is the only function with the same result type. We do not translate coercions to instance rules as we did for the sort of numbers. Instead we state that number is a subsort of numberToCall, which in GoDiS term means that any answer of the form answer(number(. . .)) is a relevant answer to the question ?x.numberToCall(x).

4 User and System Utterances in the Concrete Grammar

In this section we exemplify how it is possible to specify concrete linearizations of the abstract syntax, so that the final system can convert utterances to and from dialogue moves.

4.1 Linearizations of Dialogue Moves

In a GF grammar, each abstract function has a corresponding concrete *linearization* with the same number of arguments. E.g., the callNumber action, and the sort number, can have the following linearizations:

lin callNumber(x) = "call" ++ *variants*{x ; "a number"}
lin single(x) = x
lin cons(x,y) = x ++ y

Now, the result of parsing the sentence "call twelve nineteen sixty" will be the GF term:

callNumber(cons(12,cons(19,single(60)))) : Action(makeCall)

There is an automatic translation from GF terms to GoDiS dialogue moves, and the final result in this case will be:

request(callNumber), answer(numberToCall(cons(12,cons(19,single(60)))))

Note that there is one alternative linearization of callNumber where the argument is not used. This means that parsing of "call a number" will return callNumber(_X), which is a GF term with a metavariable _X. The translation to GoDiS dialogue moves yields:

request(callNumber), answer(numberToCall(_X))

This is equivalent to request(callNumber), since the second dialogue move is uninformative and will be ignored.

The GF linearizations are also used by the system; e.g., when it wants to raise a question or give an answer. The dialogue moves generated by GoDiS will be translated to (one or more) GF terms, which in turn are linearized to utterances. So, when the system wants to ask the question ?action(callNumber), it linearizes the term askAction(callNumber(_X)) to the resulting utterance "Do you want to call a number?".

fun askAction : (m:Menu) → Action(m) → DialogueMove
lin askAction(_)(x) = "Do you want to" ++ x ++ "?"

4.2 System Wh-questions

In the grammar, the GoDiS predicates are specified as GF categories, not grammar rules. This means that a wh-question such as ?x.numberToAdd(x) does not correspond to a GF term, but to the category numberToAdd instead. Fortunately, GF has a mechanism for specifying how to linearize unknown terms of a given category. For each GF category corresponding to a predicate we define a *linearization default*:

lindef numberToCall = "Which number do you want to call?"
lindef numberToAdd = "Which number do you want to add?"

When the GF linearizer comes across an unknown term of the category C, it uses the linearization default for C. This means that we can translate the dialogue move ask(?x.numberToAdd(x)) to a GF metavariable of type numberToAdd. GF then linearizes the metavariable to the utterance "Which number do you want to add?".

4.3 Using the GF Resource Grammar

To get more grammatically correct utterances (e.g., for congruence or different word order) we make use of complex linearization types in the GF grammar. One way to do this is to specify all grammatical parameters for the target language ourselves.

Another solution is to use the GF Resource Grammar Library for implementing the concrete syntax. The resource library is a common API for 13 languages, implemented as a large GF grammar. It can be used for writing grammatically correct domain grammars without needing perfect knowledge of the target language. Instead of writing linearization terms in the right-hand sides, we give a syntax tree from the resource grammar. As an example, the action for calling by number can be written:

lin callNumber(x) = mkVP (call_Verb)
(*variants*{x ; mkNP (a_Det) (number_Noun)})

Here, mkVP and mkNP are operations defined in the resource library, and call_Verb, a_Det and number_Noun are defined in the lexicon.

Finally, recall that a single abstract GF grammar can map to several concrete syntaxes. This can be used for writing multilingual dialogue system grammars. In particular, the GF Resource Grammar Library can be used to write a single concrete instance covering 13 languages.

5 Discussion

We have described how to give a full specification of an ISU-based dialogue system, as a GF grammar. The abstract syntax specifies the dialogue manager, and the concrete syntax specifies a mapping between GoDiS dialogue moves and natural language utterances.

Related Work. Some earlier attempts have been done on specifying dialogue systems in a single formalism. Most similar to our solution is [6], from which this article has borrowed some ideas. The advantage of our approach is that by compiling to GoDiS we get all the nice dialogue handling capabilities as exemplified in section 1.2.

Another inspiration has been [5], where some parts of a GoDiS system can be specified as an OWL ontology. The difference here is that in our system *all* necessary parts of a GoDiS system are specified in the GF grammar.

Future Work. The implementation is still only a prototype, and we plan to implement a full-scale version in the near future. A real-sized proof-of-concept dialogue system will also be implemented.

Multimodal dialogue systems as described in [11] are not currently handled, but we plan to extend the formalism to handle multiple modalities as well.

The abstract syntax of a GF grammar can be implemented as an OWL ontology [5]. We plan to explore whether it is fruitful to specify at least parts of a dialogue system in OWL.

References

1. Larsson, S.: Issue-based Dialogue Management. PhD thesis, Department of Linguistics, Gothenburg University (2002)
2. Traum, D., Larsson, S.: The information state approach to dialogue management. In Smith, Kuppevelt, eds.: Current and New Directions in Discourse and Dialogue. Kluwer Academic Publishers (2003) 325–353
3. Rayner, M., Hockey, B.A., Bouillon, P.: Putting Linguistics into Speech Recognition: The Regulus Grammar Compiler. CSLI Publications (2006)
4. Ranta, A.: Grammatical Framework, a type-theoretical grammar formalism. Journal of Functional Programming **14**(2) (2004) 145–189
5. Ljunglöf, P., Amores, G., Burden, H., Manchón, P., Pérez, G., Ranta, A.: Enhanced multimodal grammar library. Deliverable D1.5, TALK Project (August 2006)
6. Bringert, B.: Rapid development of dialogue systems by grammar compilation. In Keizer, S., Bunt, H., Paek, T., eds.: Proceedings of the 8th SIGdial Workshop on Discourse and Dialogue, Antwerp, Belgium (September 2007)
7. Martin-Löf, P.: Intuitionistic Type Theory. Bibliopolis, Napoli (1984)
8. Ranta, A., El-Dada, A., Khegai, J.: The GF Resource Grammar Library. (2006) Can be downloaded from `http://code.haskell.org/gf/doc/resource.pdf`
9. Rayner, M., Carter, D., Bouillon, P., Digalakis, V., Wirén, M.: The Spoken Language Translator. Cambridge University Press (2000)
10. Ranta, A.: Modular grammar engineering in GF. Research on Language and Computation **5**(2) (June 2007) 133–158
11. Bringert, B., Cooper, R., Ljunglöf, P., Ranta, A.: Multimodal dialogue system grammars. In: DIALOR'05, 9th Workshop on the Semantics and Pragmatics of Dialogue, Nancy, France (June 2005)