Multimodal Grammar Library

Peter Ljunglöf Gabriel Amores Robin Cooper
David Hjelm Oliver Lemon Pilar Manchón
Guillermo Pérez Aarne Ranta

Distribution: Public

TALK
Talk and Look: Tools for Ambient Linguistic Knowledge
IST-507802 Deliverable 1.2b

07/02/06

Project funded by the European Community
under the Sixth Framework Programme for
Research and Technological Development

The deliverable identification sheet is to be found on the reverse of this page.

Project ref. no.	IST-507802
Project acronym	TALK
Project full title	Talk and Look: Tools for Ambient Linguistic Knowledge
Instrument	STREP
Thematic Priority	Information Society Technologies
Start date / duration	01 January 2004 / 36 Months

Security	Public
Contractual date of delivery	Dec 05
Actual date of delivery	07/02/06
Deliverable number	1.2b
Deliverable title	Multimodal Grammar Library
Type	Report
Status & version	Public Final
Number of pages	73 (excluding front matter)
Contributing WP	1
WP/Task responsible	UGOT
Other contributors	UEDIN, USE
Author(s)	Peter Ljunglöf, Gabriel Amores, Robin Cooper, David Hjelm, Oliver Lemon, Pilar Manchón, Guillermo Pérez and Aarne Ranta
EC Project Officer	Evangelia Markidou
Keywords	grammar, multilingual, multimodal, multimodal fusion, dialogue systems, Grammatical Framework, TrindiKit, GoDiS, DelfosNCL

The partners in TALK are:

Saarland University	USAAR
University of Edinburgh HCRC	UEDIN
University of Gothenburg	UGOT
University of Cambridge	UCAM
University of Seville	USE
Deutches Forschungszentrum fur Künstliche Intelligenz	DFKI
Linguamatics	LING
BMW Forschung und Technik GmbH	BMW
Robert Bosch GmbH	BOSCH

For copies of reports, updates on project activities and other TALK-related information, contact:

The TALK Project Co-ordinator
Prof. Manfred Pinkal
Computerlinguistik
Fachrichtung 4.7 Allgemeine Linguistik
Postfach 15 11 50
66041 Saarbrücken, Germany
pinkal@coli.uni-sb.de
Phone +49 (681) 302-4343 - Fax +49 (681) 302-4351

Copies of reports and other material can also be accessed via the project's administration homepage, http://www.talk-project.org

Contents

Summary

The ISU approach uses abstract representations for dialogue states and update rules which allow the generic characterisation of flexible dialogue strategies. This enables the same code for dialogue management techniques to be used for different natural languages and for different domains.

In this deliverable, we show that by using an abstract representation for grammars, we can further enable rapid porting of dialogue systems between languages, domains and modalities. The main tool in defining such grammars is Grammatical Framework (GF), which is used in collaboration by UGOT, UEDIN and UCAM for making ISU-based dialogue systems.

We describe two approaches to adding multimodality to unimodal dialogue systems and grammars. The first approach is to implement multimodality at the grammar level. We give a language- and domain-independent method for how to add multimodal information to a unimodal GF grammar, thus simplifying the transition from a speech-only dialogue system to a multimodal one. The second approach is to implement multimodality at the level of the dialogue manager, which is tried out in the ISU-based dialogue system DelfosNCL, developed by USEV.

The main part of the deliverable is a detailed description of the multimodal and multilingual GF/GoDiS grammar library, written in Grammatical Framework. The grammar library connects user and system utterances specified in GF with a dialogue system using the ISU-based GoDiS dialogue manager. The library is designed for making it easy to add new dialogue domains, source languages, and input and output modalities. Currently the library consists of two dialogue domains, each with two source languages and three different modalities. The two domains are the calendar application *AgendaTalk*, and the MP3 player *DJ GoDiS*.

Furthermore, two additional multimodal GF grammars are described, which have been created using the method for adding multimodality. They are not part of the GF/GoDiS grammar library since they are not part of a GoDiS dialogue system, but can be seen as proofs-of-concept of the generality of the method. The *Tram Demo* grammar, used by the UGOT Tram Information System (GOTTIS), is used as a pedagogical example when introducing the method. The UEDIN *Town Info* grammar, used with the ISU-based DIPPER dialogue manager, is used to test the method.

By using the diversity of the GF module system, such as resource modules, incomplete modules, interfaces and instances, we have maximized sharing of common information between languages, modalities, ontologies and domains. This is done to make adding a new language, modality, ontology or domain as simple as possible.

Chapter 1

Introduction

This deliverable concerns the development of technology incorporating multimodality into dialogue systems using grammars. We discuss two alternative strategies: multimodality can either be handled as part of dialogue management or we can construct multimodal grammars. We define a general way of creating multimodal grammars from unimodal grammars and describe the multimodal grammar library that has been built using these techniques.

In making the Multimodal Grammar Library we exploit the advantages of the ISU approach. The ISU approach utilizes structured Information States to keep track of dialogue context information. These Information States can be read and updated by several different modules which access precisely the information that they need. This enables a modular architecture which allows generic solutions for dialogue technology. For example,

- different language modules can interact with essentially similar Information States, enabling rapid porting of dialogue systems from one language to another and the creation of multilingual dialogue systems;
- coding of dialogue behaviour is supported independently of language and domain, thus allowing for the rapid porting of dialogue systems to different domains;
- the use of structured Information States allows straightforward implementation of flexible dialogue systems which can access and modify information in the Information State in different sequences and by varying means.

In this deliverable, as well as in the earlier deliverables D1.1 and D1.2a [Ljunglöf et al., 2005, Bringert et al., 2005], we show that by using an abstract representation for grammars, we can further enable rapid porting of dialogue systems between languages, domains and modalities. The main tool in defining such grammars is Grammatical Framework (GF), which is used in collaboration by UGOT, UEDIN and UCAM for making ISU-based dialogue systems.

Layout of the deliverable

We begin by giving a general description of multimodal interfaces and make a distinction between *parallel* and *integrated multimodality* (the latter is also known as *multimodal fusion*). We then give an introduction to the two grammar systems we have been working with – Grammatical Framework (GF) and DelfosNCL.

In chapter 2 we present techniques for incorporating multimodality into using these two grammar systems. We describe how multimodality can be specified in a GF grammar, and give a method for adding multimodality to a unimodal grammar. As a pedagogical example we describe a multimodal version of the UGOT *Tram Demo* grammar. Furthermore, we describe and compare two strategies for multimodal fusion in DelfosNCL.

In chapter 3 we describe the contents of the multimodal grammar library, which is implemented in GF as a front-end to the generic dialogue system GoDiS built within TrindiKit. The library also includes several ontology databases and two example dialogue domains – a calendar application called *AgendaTalk*, and an MP3 player application called *DJ GoDiS*. The library is designed for making is easy to add new dialogue domains, source languages, and input and output modalities. As a test of the generality of the method of adding multimodality to a grammar, the multimodal version of the UEDIN *Town Info* grammar is described.

1.1 Multimodal Interfaces

Multimodal interfaces allow for more flexible and natural interactions between human users and computer systems. They benefit from a variety of communication channels such as speech, text, gesture, handwriting, etc. Multimodal systems have been largely studied since the appearance of the "Put-That-There" system [Bolt, 1980]. The results of Oviatt et al. [1997] showed the potential benefits of multimodal systems compared to unimodal ones in terms of user preferences and the possibility of mutual disambiguation.

The fusion of multimodal inputs has also evolved since Bolt's proposal, which suffered from lack of generality, defining rules that could only apply to speech-driven systems. Johnston [1998] proposed a new approach using a unification based multidimensional parsing of typed feature structures that partially overcame the limitations previously mentioned. Johnston and Bangalore [2000] found that this solution could be improved both at parsing level, because of its inherent computational complexity, and at natural language understanding level because it did not allow a tight-coupling of parsing and input recognition (speech or gesture). They proposed an alternative approach using finite-state multimodal grammars.

In this deliverable we explore two ways of implementing multimodal interfaces. In the first approach we follow in Johnston's footsteps and implement multimodality at the grammar-level, as multimodal GF grammars. As we will show, GF is well suited for implementing multimodality – different modalities can be realised as discontinuous constituents, and the possibility of defining macros in GF can be used to streamline the task of adding multimodality to a unimodal grammar. Also, there are efficient parsing algorithms for GF grammars with discontinuous constituents [Ljunglöf, 2004, Burden and Ljunglöf, 2005].

In the second approach we implement multimodal fusion not at grammar level, but at the level of the dialogue manager within an Information State Update (ISU) approach. This approach is tried out in DelfosNCL.

1.1.1 Parallel multimodality

Parallel multimodality is a straightforward instance of multilinguality. It means that the concrete syntaxes associated with an abstract syntax are not just different natural languages, but different representation modalities, encoded by language-like notations such as graphic representation formalisms. Examples of

parallel multimodality are:

- When a route is described, in parallel, by speech and by a line drawn on a map.
- When a list of objects is presented, in parallel, by speech and as a list of the computer screen.

Both descriptions convey the full information alone, without support from the other. This raises the dialogue management issue of whether all information should be presented in all modalities. For the above given examples:

- All stops are indicated on the graphical presentation of a route, whereas in the natural language presentation only stops where the user must change are presented.
- If the list of objects is large, it is unfeasible to present all of them in natural language. An alternative is to say e.g. how many objects there are and just name the first. In the graphical representation however, the full list of objects can be presented.

Because GF permits the suppression of information in concrete syntax, this issue can be treated on the level of grammar instead of dialogue management.

1.1.2 Integrated multimodality

Demonstrative expressions are an old idea, which provide an example of *integrated multimodality*, as opposed to parallel multimodality. In parallel multimodality, speech and other modes of communication are just alternative ways to convey the same information. Demonstrative expressions, however, get their meaning from the context:

> *This train* is faster than *that airplane*.
>
> I want to go from *this place* to *this place*.
>
> I would like to listen to *this song*.

In particular, as in these examples, the meaning can be obtained from accompanying pointing gestures.

Thus the meaning-bearing unit is neither the words nor the gestures alone, but the combination of the two modalities. The problem of multimodal fusion described above is then how to combine the two modalities into one multimodal utterance. How to define integrated multimodality with a grammar is less obvious than parallel multimodality. The GF solution makes essential use of records, and not just strings, as outcomes of linearization. In brief, different modality "channels" are stored in different fields of a record, and it is the combination of the different fields that is sent to the dialogue system parser. The DelfosNCL solution is to implement fusion at the dialogue level instead of at the grammar level.

Representing demonstratives in semantics and grammar

When formalizing the semantics of demonstratives, we can combine syntax with coordinates:

> I want to go from *this place* to *this place*.

is interpreted as something like

```
want(I, go, this(place,(123,45)), this(place,(98,10)))
```

Now, the same semantic value can be given in many ways, by performing the clicks at different points of time in relation to the speech:

I want to go from *this place* CLICK(123,45) to *this place* CLICK(98,10)

I want to go from *this place* to *this place* CLICK(123,45) CLICK(98,10)

CLICK(123,45) CLICK(98,10) I want to go from *this place* to *this place*

How do we build the value compositionally in parsing? Traditional parsing is sequential: its input is a string of tokens. It works for demonstratives only if the pointing is adjacent to the spoken expression. In the actual input, the demonstrative word can be separated from the accompanying click by other words. The two can also be simultaneous.

1.2 Grammatical Framework

In this section we only describe some details of Grammatical Framework (GF) which are crucial for this deliverable. GF is described in more detail by Ranta [2004], in TALK deliverable D1.2a [Bringert et al., 2005], and on the GF homepage:

```
http://www.cs.chalmers.se/~aarne/GF/
```

1.2.1 Multimodal dialogue system grammars

Asynchronous syntax in GF

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:

abstract syntax trees $\Longleftrightarrow$ concrete syntax objects

When modelling context-free grammar in GF, the concrete syntax objects are just strings. But they can be more structured objects as well – in general, they are *records* of different kinds of objects. For example, a demonstrative expression can be linearized into a record of two strings:

this place CLICK(123,45) $\Longleftrightarrow$ `{s = "this place"; p = "(123,45)"}`

The record

```
{s = "I want to go from this place to this place";
 p = "(123,45) (98,10)"}
```

represents any combination of the sentence and the clicks, as long as the clicks appear in this order, including the examples at the end of section 1.1.2.

Integrated multimodality and discontinuous constituents

The GF representation of integrated multimodality is similar to the representation of *discontinous constituents*. For instance, assume *"has arrived"* is a verb phrase in English, which can be used both in declarative sentences and questions,

she *has arrived*

has she *arrived*

In the question, the two words are separated from each other. If *"has arrived"* is a constituent of the question, it is thus discontinuous. To represent such constituents in GF, records can be used: we split verb phrases (`VP`) into a finite and infinitive part.

```
lincat VP = {fin, inf : Str};

lin Indic np vp = {s = np.s ++ vp.fin ++ vp.inf};
    Quest np vp = {s = vp.fin ++ np.s ++ vp.inf};
```

From grammars to dialogue systems

The general recipe for using GF when building dialogue systems is to write a grammar with the following components:

- The abstract syntax defines the semantics (the "ontology") of the domain of the system.
- The concrete syntaxes define alternative modes of input and output.

The engineering advantages of this approach have to do partly with the declarativity of the description, partly with the tools provided by GF to derive different components of the system:

- The type checker guarantees that all the input and output modes match with the ontology.
- The grammar compiler generates parsers for each input grammar and generators for each output grammar.
- Translators between GF's abstract syntax and other ontology description languages enable communication with different kinds of dialogue managers and cover e.g. Prolog terms and XML objects.
- Translators from GF's concrete syntax to speech recognition formats make it possible to generate e.g. Nuance grammars and ATK language models.

1.2.2 Resource modules – reusing common information

Apart from abstract and concrete modules, there is a third kind of grammar module in GF called *resource modules*. There are two kinds of judgements possible in a resource module: parameter declarations, and operator definitions.

The parameters and operators in a resource module are imported to a concrete grammar by *opening* the resource module:

```
concrete Cnc of Abs = open Res in ...
```

Several resource modules can be opened in parallel:

```
concrete Cnc of Abs = open Res_1, ..., Res_n in ...
```

If a parameter `P` or operation `F` is defined in several resource modules, there is a conflict and we have to use qualified reference $\texttt{Res}_i\texttt{.P}$ or $\texttt{Res}_i\texttt{.F}$ to disambiguate.
We can also introduce abbreviations for writing `R.P` or `S.F` instead of $\texttt{Res}_1\texttt{.P}$ or $\texttt{Res}_n\texttt{.F}$:

```
concrete Cnc of Abs = open (R=Res_1), ..., (S=Res_n) in ...
```

Parameter declarations

Parameters are non-recursive datatypes which are used in concrete linearizations when describing inflection tables, and inherent feaures. Standard examples are the source-language specific parameters *number*, *gender*, *case* etc.:[1]

```
param Number = Sing  | Plur;
      Gender = Neutr | Utr Masc;
      Masc   = Masc  | Nomasc;
      Case   = Nom   | Gen;
```

Note that parameters can be hierarchic, as in the definition of Swedish gender. But they are not allowed to be recursive, meaning that there are always a finite number of parameters of a given parameter type.
Parameters are used in inflection tables and as inherent features, and are further described in TALK deliverable D1.2a [Bringert et al., 2005], and by Ranta [2004].

Operation definitions

Operations in GF are defined in a rich functional language, and are always typed. Dependent types and higher-order functions may be used. The main restriction is that the operation definition must not be recursive, which together with some further minor restrictions means that whenever they are used in a concrete grammar module, they can be compiled away. Thus, the posibility of defining operations can be seen as a very expressive macro facility.
An operation definition consists of a typing and a defining expression:

```
oper f : T_1 -> ... -> T_n -> T
       = \x_1,...,x_n -> t;
```

The typing restriction says that t must be an expression of type T whenever x_i is of type T_i ($1 \leq i \leq n$). Dependent types are allowed, as well as the top-level type `Type`, so we can write general operations that works for any type T:

[1] These are Swedish parameter definitions – other languages have other parameters, and other definitions. E.g. German have three genders and four cases.

```
oper AddStr : Type -> Type
            = \t -> t ** {s : Str};
     addStr : (t:Type) -> Str -> t -> AddStr t
            = \_,str,term = term ** {s = str};
```

The first operation `AddStr` creates a new linearization type by adding a record row `{s:Str}`; and the second operation `addStr` adds a given string to a term of type `t`, returning an object of type `AddStr t`. Note the built-in record extension operator (`**`), which extends a record type or a record with new rows.

Using abstract and concrete grammars as resources

There is a built-in translation of an abstract with a corresponding concrete grammar module into a resource module. Each abstract category *C* is translated to an operation *C* of type `Type`, with the linearization type as its definition:[2]

```
cat C;          ⟹    oper C : Type = T;
lincat C = T;
```

Each abstract function f of category *C* with linearization *t* is translated to an operation f of type *C* with definition *t*:

```
fun f : C1 -> ... -> Cn -> C;   ⟹   oper f : C1 -> ... -> Cn -> C
lin f x1 ... xn = t;                        = \x1,...,xn -> t;
```

This translation means that we can open a concrete module as well as a resource module when defining a new concrete grammar module:

```
concrete NewCnc of Abs = open OldCnc in ...
```

This will be used in the multimodal grammar library in chapter 3, to extend a unimodal grammar with multimodal information. In that case both concrete grammars also share the same abstract syntax, meaning that each function f already has a definition in `OldCnc`, which we can extend with some extra information:

```
lin f x1 ... xn = OldCnc.f x1 ... xn ** (new information);
```

Interfaces, instances and incomplete grammars

Operation definitions can be split into the typing and the expression separately:

```
oper f : T1 -> ... -> Tn -> T;
oper f x1 ... xn = t;
```

[2] This translation is a slight simplification since we do not mention the nullary *lock fields* which are automatically added to the translation for making type checking correct.

Note that in this case, the lambda abstraction $\backslash x_1, \ldots, x_n$ can be moved to the left-hand side of the definition.

This makes it possible to put all typings in a separate module, which is called an interface module:

```
interface ResI = {
  oper hello : Str -> {s : Str};
}
```

This interface can now be instantiated in different source languages by instance modules:

```
instance ResEng of ResI = {
  oper hello name = {s = "Hello" ++ name ++ "!"};
}
instance ResSwe of ResI = {
  oper hello name = {s = "Hejsan" ++ name ++ "!"};
}
```

An instance module is equivalent to a resource module, and can be opened by a concrete grammar in the same way. But an interface module can also be opened by a concrete grammar, which then becomes *incomplete*:

```
incomplete concrete CncI of Abs = open ResI in {
  lin greeting = hello "Dolly";
}
```

An incomplete module can be completed by instantiating all opened interfaces:

```
concrete CncEng of Abs = CncI with (ResI=ResEng);
```

These features are also used in the grammar library in chapter 3 for increasing sharing between grammars.

1.2.3 Hierarchical grammar modules

The GF/GoDiS grammar library described in chapter 3 consists of a quite large number of files. To make the module structure more explicit we have used a hierarchical module structure, where the hierarchy is reflected in the file structure. However, hierarchical modules is not implemented in GF version 2.4.[3]

Therefore we have chosen to name the modules as follows. A hierarchical GF module is named `A_B_C` and is physically located in the file `A_B_C.gf` residing in the directory `A/B`. This means that the module hierarchy reflects the physical directory structure of the grammar files. With this solution, hierarchical modules can be used in the current version of GF. The drawback is that `A` and `B` have to be repeated in both the file name and the directory structure. When hierarchical modules is supported by GF, we can drop `A` and `B` from the file name to get `C.gf`. We can also drop all search paths for impoerted modules, which currently have to be included in the grammar files.

[3] GF 2.4 is the current public version as of 31st January 2006.

1.3 DelfosNCL

DelfosNCL can be described as a collaborative dialogue manager linked to a Natural Language Understanding Module, which allows dialogues driven by the semantic information provided by the user and by the dialogue expectations generated by the dialogue manager. The kernel of our system is then composed by two main modules:

- A Natural Language Understanding (NLU) module which is in charge of the lexical and syntactic analysis and produces the Information States, and
- A Dialogue Manager which manipulates Information States (or Dialogue Moves) through the application of dialogue update rules

The Information States configured for this scenario are based on the DTAC protocol [Quesada et al., 2000], A DTAC consists of a feature-value structure with four main features: DMOVE, TYPE, ARG and CONT. The following figure illustrates the DTAC obtained for the command *"Turn on the kitchen light"* in our scenario:

$$\begin{bmatrix} \text{DMOVE} & \text{specifyCommand} \\ \text{TYPE} & \text{switchOn} \\ \text{ARG} & \text{Device} \\ \text{DEVICE} & \begin{bmatrix} \text{DMOVE} & \text{specifyParameter} \\ \text{TYPE} & \text{OnOffDevice} \\ \text{CONT} & \text{kitchen} \end{bmatrix} \end{bmatrix}$$

Dialogue Update Rules take the following form in our system:

```
( RuleID: MAKECALL;
    PriorityLevel: 15;
    TriggeringCondition: (DMOVE:specifyCommand,TYPE:MakeCall);
    DeclareExpectations: {
         Dest <= (DMOVE:specifyParameter,TYPE:Name|PhoneNumber); }
    SetExpectations: {
        Confirm <= (DMOVE:answerYN); }
    ActionsExpectations:
        {   [Dest] => {ExecuteDMFunction(MakeCallDest); }
            [Confirm] => {ExecuteDMFunction(MakeCallDisam); }
    PostActions: { @if(@is-MAKECALL.Confirm.TYPE == "YES") {
                     ExecuteDMFunction(MakeCallDest);
                     }
                 }
    }
)
```

The item **TriggeringCondition** describes the Dialogue Move (DMove) that must arrive for the rule to be activated. **DeclareExpectations** defines additional information needed for the rule to be fulfilled. This information could have been provided previously in the dialogue history, or during the same interaction.

The **SetExpectations** section defines additional Dialogue Moves (DMoves) needed to successfully execute the rule, such as an explicit confirmation before executing a command.

As its name indicates, **ActionExpectations** defines the actions to be carried out when either the **DeclareExpectations** have not been fulfilled by the current input nor within the Dialogue History, or when some **SetExpectations** have been defined.

Finally, the **PostActions** section describes what should be done once the rule is active and all the expectations have been fulfilled.

1.4 Summary

In this chapter we have introduced the different kinds of multimodal interfaces and the different systems we are focussing on.

Multimodal interfaces can be split into parallel multimodality and integrated multimodality. Parallel multimodality is when the same information is displayed simultaneously in several modalities. Integrated multimodality is when different modalities convey different parts of the information, and the problem of multimodal fusion is then how to combine the information from the different modalities into one single utterance.

The systems we are focussing on are Grammatical Framework (GF) and DelfosNCL. GF is a grammar formalism and an implementation, well suited for describing multilinguality and multimodality, as already shown in TALK deliverables 1.1 and 1.2a [Ljunglöf et al., 2005, Bringert et al., 2005]. In this deliverable we show the multimodal capabilities in more detail by giving a multimodal and multilingual grammar library for several dialogue applications. DelfosNCL is a ISU-based multimodal dialogue system composed of a natural language understanding module which analyses the input and produces information states, and a dialogue manager which manipulates the information states.

Chapter 2

Multimodal grammars and multimodal fusion

This chapter is divided into two parts. Section 2.1 describes how multimodality can be specified in Grammatical Framework, and gives a method for how to add multimodality to a unimodal grammar. As a pedagogical example we describe a multimodal version of the UGOT *Tram Demo* grammar. Section 2.2 describes two strategies for multimodal fusion in DelfosNCL, and compares the advantages and disadvantages.

2.1 Demonstrative Expressions and Multimodal Grammars

This section shows a method to write GF grammars in which spoken utterances are accompanied by pointing gestures. The method is introduced via a concrete example how multimodal grammars can be written in GF and how they can be used in dialogue systems. The explanation is given in three stages:

1. How to write a multimodal grammar by hand.
2. How to add multimodality to a unimodal grammar.
3. How to use a multimodal resource grammar.

Example multimodal grammar: abstract syntax

A simple example of a multimodal GF grammar is the one called the *Tram Demo* grammar, which is described in more detail in TALK deliverable D1.2a [Bringert et al., 2005]. The grammar is a part of a dialogue system that deals with queries about tram timetables. The system interprets a speech input in combination with mouse clicks on a digital map.
The abstract syntax of (a minimal fragment of) the *Tram Demo* grammar is

```
cat Input, Dep, Dest, Click;
fun GoFromTo    : Dep  -> Dest -> Input; -- "I want to go from x to y"
    DepHere     : Click -> Dep;          -- "from here" with click
    DestHere    : Click -> Dest;         -- "to here" with click
```

```
fun CCoord      : Int -> Int -> Click;   -- click coordinates
```

An English concrete syntax of the grammar is

```
lincat Input, Dep, Dest = {s : Str; p : Str};
       Click            = {p : Str};

lin GoFromTo x y  = {s = ["I want to go"] ++ x.s ++ y.s; p = x.p ++ y.p};
    DepHere c     = {s = ["from here"]                ; p = c.p};
    DestHere c    = {s = ["to here"]                  ; p = c.p};

lin CCoord x y    = {p = "(" ++ x.s ++ "," ++ y.s ++ ")"};
```

When the grammar is used in the actual system, standard parsing methods are used for interpreting the integrated speech and click input. Parsing appears on two levels: the speech input parsing performed by the Nuance speech recognition program (without the clicks), and the semantics-yielding parser sending input to the dialogue manager. The latter parser just attaches the clicks to the speech input. The order of the clicks is preserved, and the parser can hence associate each of the clicks with proper demonstratives. Here is the grammar used in the two parsing phases.

```
cat Query,   -- whole content
    Speech;  -- speech only
fun QueryInput  : Input -> Query;   -- the whole content shown
    SpeechInput : Input -> Speech;  -- only the speech shown

lincat Query, Speech = {s : Str};
lin QueryInput  i = {s = i.s ++ ";" ++ i.p};
    SpeechInput i = {s = i.s};
```

2.1.1 Adding multimodality to a unimodal grammar

This section gives a recipe for making any unimodal grammar multimodal, by adding pointing gestures to chosen expressions. The recipe guarantees that the resulting grammar remains semantically well-formed, i.e. type correct.

The multimodal conversion

The *multimodal conversion* of a grammar consists of seven steps, of which the first is always the same, the second involves a decision, and the rest are derivative:

1. Add the category `Point` with a standard linearization type.

   ```
   cat Point;
   lincat Point = {point : Str};
   ```

2. (Decision) Decide which constructors are demonstrative, i.e. take a pointing gesture as an argument. Add a `Point` as their last argument. The new type signatures for such constructors *d* have the form

   ```
   fun d : ... -> Point -> D;
   ```

3. (Derivative) Add a `point` field to the linearization type *L* of any demonstrative category *D*, i.e. a category that has at least one demonstrative constructor:

   ```
   lincat D = L ** {point : Str};
   ```

4. (Derivative) If some other category *C* has a constructor *d* that takes demonstratives as arguments, make it demonstrative by adding a *point* field to its linearization type.

5. (Derivative) Store the `point` field in the linearization *t* of any constructor *d* that has been made demonstrative:

   ```
   lin d x1 ... xn p = t x1 ... xn ** {point = p.point};
   ```

6. (Derivative) For each constructor *f* that takes demonstratives $D_1, \ldots, D_n$ as arguments, collect the *point* fields of the arguments in the *point* field of the value:

   ```
   lin f x1 ... xm =
       t x1 ... xm ** {point = xd1.point ++ ... ++ xdn.point};
   ```

   Make sure that the pointings $\texttt{x}d_1\texttt{.point} \ldots \texttt{x}d_n\texttt{.point}$ are concatenated in the same order as the arguments appear in the *linearization t*, which is not necessarily the same as the abstract argument order.

7. (Derivative) To preserve type correctness, add an empty `point` field to the linearization *t* of any constructor *c* of a demonstrative category:

   ```
   lin c x1 ... xn = t x1 ... xn ** {point = []};
   ```

An example of the conversion

Start with a *Tram Demo* grammar with no demonstratives, but just tram stop names and the indexical *here* (interpreted as e.g. the user's standing place).

```
cat Input, Dep, Dest, Name;
fun GoFromTo    : Dep  -> Dest -> Input;
    DepHere     : Dep;
    DestHere    : Dest;
    DepName     : Name -> Dep;
    DestName    : Name -> Dest;
    Almedal     : Name;
```

A unimodal English concrete syntax of the grammar is

```
lincat Input, Dep, Dest, Name = {s : Str};
lin GoFromTo x y  = {s = ["I want to go"] ++ x.s ++ y.s};
    DepHere       = {s = ["from here"]};
    DestHere      = {s = ["to here"]};
    DepName n     = {s = ["from"] ++ n.s};
    DestName n    = {s = ["to"] ++ n.s};
    Almedal       = {s = "Almedal"};
```

Let us follow the steps of the recipe.

1. We add the category `Point` and its linearization type.
2. We decide that `DepHere` and `DestHere` involve a pointing gesture.
3. We add `point` to the linearization types of `Dep` and `Dest`.
4. Therefore, also add `point` to `Input`. (But `Name` remains unimodal.)
5. Add `p.point` to the linearizations of `DepHere` and `DestHere`.
6. Concatenate the points of the arguments of `GoFromTo`.
7. Add an empty `point` to `DepName` and `DestName`.

In the resulting grammar, one category is added and two functions are changed in the abstract syntax (annotated by the step numbers):

```
cat Point;                                          -- 1
fun DepHere     : Point -> Dep;                     -- 2
    DestHere    : Point -> Dest;                    -- 2
```

The concrete syntax in its entirety looks as follows

```
lincat Dep, Dest = {s : Str; point : Str};          -- 3
       Input = {s : Str; point : Str};              -- 4
       Name = {s : Str};
       Point = {point : Str};                       -- 1

lin GoFromTo x y  = {s = ["I want to go"] ++ x.s ++ y.s; -- 6
                     point = x.point ++ y.point};
    DepHere p     = {s = ["from here"];             -- 5
                     point = p.point};
    DestHere p    = {s = ["to here"];               -- 5
                     point = p.point};
    DepName n     = {s = ["from"] ++ n.s;           -- 7
                     point = []};
    DestName n    = {s = ["to"] ++ n.s;             -- 7
                     point = []};
    Almedal       = {s = "Almedal"};
```

What we need in addition, to use the grammar in applications, are

1. Constructors for `Point`, e.g. coordinate pairs.
2. Top-level categories, like `Query` and `Speech` in the original.

But their proper place is probably in another grammar module, so that the core *Tram Demo* grammar can be used in different systems e.g. encoding clicks in different ways.

Multimodal conversion combinators

GF is a functional programming language, and we exploit this by providing a set of combinators that makes the multimodal conversion easier and clearer. We start with the type of sequences of pointing gestures.

```
oper Point : Type = {point : Str};
```

To make a record type multimodal is to extend it with `Point`. The record extension operator `**` is needed here.

```
oper Dem   : Type -> Type = \t -> t ** Point;
```

To construct, use, and concatenate pointings:

```
oper mkPoint : Str -> Point = \s -> {point = s};
     noPoint : Point = mkPoint [];
     point   : Point -> Str = \p -> p.point;
     concatPoint : Point -> Point -> Point = \x,y ->
                   mkPoint (point x ++ point y);
```

Finally, to add pointing to a record, with the limiting case of no demonstrative needed.

```
oper mkDem  : (t : Type) -> t -> Point -> Dem t = \_,x,s -> x ** s;
     nonDem : (t : Type) -> t -> Dem t = \t,x -> mkDem t x noPoint;
```

Let us rewrite the *Tram Demo* grammar by using these combinators:

```
oper SS : Type = {s : Str};

lincat Input, Dep, Dest = Dem SS;
       Name = SS;

lin GoFromTo x y  = {s = ["I want to go"] ++ x.s ++ y.s} **
                    concatPoint x y;
    DepHere       = mkDem  SS {s = ["from here"]};
    DestHere      = mkDem  SS {s = ["to here"]};
    DepName n     = nonDem SS {s = ["from"] ++ n.s};
    DestName n    = nonDem SS {s = ["to"] ++ n.s};
    Almedal       = {s = "Almedal"};
```

The type synonym `SS` is introduced to make the combinator applications concise. Notice the use of partial application in `DepHere` and `DestHere`; an equivalent way to write is

```
lin DepHere p = mkDem SS {s = ["from here"]} p;
```

2.1.2 Multimodal resource grammars

The main advantage of using GF when building dialogue systems is that various components of the system can be automatically generated from GF grammars. Writing these grammars, however, can still be a considerable task. A case in point are multilingual systems: how to localize e.g. a system built in a car to the languages of all those customers to whom the car is sold? This problem has been the main focus of GF for some years, and the solution on which most work has been done is the development of *resource grammar libraries*. These libraries work in the same way as program libraries in software engineering, enabling a division of labour between linguists and domain experts.

One of the goals in the resource grammars of different languages has been to provide a *language-independent API*, which makes the same resource grammar functions available for different languages. For instance, the categories `S`, `NP`, and `VP` are available in all of the 10 languages currently supported, and so is the function

```
PredVP : NP -> VP -> S
```

which corresponds to the rule **S** → **NP VP** in phrase structure grammar. However, there are several levels of abstraction between the function `PredVP` and the phrase structure rule, because the rule is implemented in so different ways in different languages. In particular, discontinuous constituents are needed in various degrees to make the rule work in different languages.

Now, dealing with discontinuous constituents is one of the demanding aspects of multilingual grammar writing that the resource grammar API is designed to hide. But the proposed treatment of integrated multimodality is heavily dependent on similar things. What can we do to make multimodal grammars easier to write (for different languages)? There are two orthogonal answers:

1. Use resource grammars to write a unimodal dialogue grammar and then apply the multimodal conversion to manually chosen parts.

2. Use *multimodal resource grammars* to derive multimodal dialogue system grammars directly.

The multimodal resource grammar library has been obtained from the unimodal one by applying the multimodal conversion manually. In addition, the API has been simplified by leaving out structures needed in written technical documents (the original application area of GF) but not in spoken dialogue.

In the following subsections, we will show a part of the multimodal resource grammar API, limited to a fragment that is needed to get the main ideas and to reimplement the *Tram Demo* grammar. The reimplementation shows one more advantage of the resource grammar approach: dialogue systems can be automatically instantiated to different languages.

Resource grammar API

The resource grammar API has three main kinds of entries:

1. Language-independent linguistic structures ("linguistic ontology"), e.g.

```
PredVP : NP -> VP -> S;      -- "Mary helps him"
```

2. Language-specific syntax extensions, e.g. Swedish and German fronting topicalization

```
TopicObj : NP -> VP -> S;    -- "honom hjälper Mary"
```

3. Language-specific lexical constructors, e.g. Germanic *Ablaut* patterns

```
irregV : (sing,sang,sung : Str) -> V;
```

The first two kinds of entries are `cat` and `fun` definitions in an abstract syntax. The multimodal, restricted API has e.g. the following categories. Their names are obtained from the corresponding unimodal categories by prefixing `M`.

```
cat MS;      -- multimodal sentence or question
    MQS;     -- multimodal wh question
    MImp;    -- multimodal imperative
    MVP;     -- multimodal verb phrase
    MNP;     -- multimodal (demonstrative) noun phrase
    MAdv;    -- multimodal (demonstrative) adverbial

cat Point;   -- pointing gesture
```

Multimodal API: functions for building demonstratives

Demonstrative pronouns can be used both as noun phrases and as determiners.

```
fun this_MNP    : Point -> MNP;          -- this
    thisDet_MNP : CN -> Point -> MNP;    -- this car
```

There are also demonstrative adverbs, and prepositions give a productive way to build more adverbs.

```
fun here_MAdv      : Point -> MAdv;       -- here
    here7from_MAdv : Point -> MAdv;       -- from here
    MPrepNP        : Prep -> MNP -> MAdv; -- in this car
```

Multimodal API: functions for building sentences and phrases

A handful of predication rules construct sentences, questions, and imperatives.

```
fun MPredVP   : MNP -> MVP -> MS;     -- this plane flies here
    MQPredVP  : MNP -> MVP -> MQS;    -- does this plane fly here
    MQuestVP  : IP  -> MVP -> MQS;    -- who flies here
    MImpVP    : MVP -> MImp;          -- fly here!
```

Verb phrases are constructed from verbs (inherited as such from the unimodal API) by providing their complements.

```
fun MUseV     : V   -> MVP;           -- flies
    MComplV2  : V2  -> MNP -> MVP;    -- takes this
    MComplVV  : VV  -> MVP -> MVP;    -- wants to take this
```

A multimodal adverb can be attached to a verb phrase.

```
fun MAdvVP    : MVP -> MAdv -> MVP;   -- flies here
```

Language-independent implementation: examples

The implementation makes heavy use of the multimodal conversion combinators. It adds a `point` field to whatever the implementation of the unimodal category is in any language. Thus, for example

```
lincat MVP   = Dem VP;
       MNP   = Dem NP;
       MAdv  = Dem Adv;
lin this_MNP          = mkDem NP this_NP;
    MComplV2 verb obj = mkDem VP (ComplV2 verb obj) obj;
    MAdvVP   vp   adv = mkDem VP (AdvVP vp adv) (concatPoint vp adv);
```

Note that `mkDem` makes the definition of `this_MPN` equivalent to

```
lin this_MNP p = this_NP ** {point = p.point};
```

Multimodal API: interface to unimodal expressions

Using nondemonstrative expressions as demonstratives:

```
fun DemNP   : NP  -> MNP;
    DemAdv  : Adv -> MAdv;
```

Building top-level phrases:

```
fun PhrMS   : Pol -> MS   -> Phr;
    PhrMS   : Pol -> MS   -> Phr;
    PhrMQS  : Pol -> MQS  -> Phr;
    PhrMImp : Pol -> MImp -> Phr;
```

Instantiating multimodality to different languages

The implementation above has only used the resource grammar API, not the concrete implementations. The library `Demonstrative` is a *parametrized module*, also called a *functor*, which has the following structure

```
incomplete concrete DemonstrativeI of Demonstrative =
    Cat, TenseX ** open Test, Structural in ...
```

It can be *instantiated* to different languages as follows.

```
concrete DemonstrativeEng of Demonstrative =
    CatEng, TenseX ** DemonstrativeI with
      (Test = TestEng),
      (Structural = StructuralEng);

concrete DemonstrativeSwe of Demonstrative =
    CatSwe, TenseX ** DemonstrativeI with
      (Test = TestSwe),
      (Structural = StructuralSwe);
```

Language-independent reimplementation of Tram Demo

Again using the functor idea, we reimplement `TramDemo` as follows:

```
incomplete concrete TramI of Tram = open Multimodal in {

lincat Query = Phr; Input = MS;
       Dep, Dest = MAdv; Click = Point;

lin QInput = PhrMS PPos;
    GoFromTo x y =
       MPredVP (DemNP (UsePron i_Pron))
         (MAdvVP (MAdvVP (MComplVV want_VV (MUseV go_V)) x) y);
    DepHere    = here7from_MAdv;
    DestHere   = here7to_MAdv;
    DepName s  = MPrepNP from_Prep (DemNP (UsePN (SymbPN (MkSymb s))));
    DestName s = MPrepNP to_Prep (DemNP (UsePN (SymbPN (MkSymb s))));
}
```

Then we can instantiate this to all languages for which the `Multimodal` API has been implemented:

```
concrete TramEng of Tram = TramI with
    (Multimodal = MultimodalEng);

concrete TramSwe of Tram = TramI with
    (Multimodal = MultimodalSwe);
```

The order problem

It was pointed out in the section on the multimodal conversion that the concrete word order may be different from the abstract one, and vary between different languages. For instance, Swedish topicalization

> Det här tåget vill den här kunden inte ta.
> (“this train, this customer doesn’t want to take”)

may well have an abstract syntax of a form in which the customer appears before the train.

This is a problem for the implementor of the resource grammar. It means that some parts of the resource must be written manually and not as a functor. However, the *user* of the resource can safely ignore the word order problem, if it is correctly dealt with in the resource.

A recipe for using the resource library

When starting to develop resource grammars, we believed they would be all that an application grammarian needs to write a concrete syntax. However, experience has shown that it can be tough to start grammar development in this way: selecting functions from a resource API requires more abstract thinking than just writing strings, and its take longer to reach testable results. The most light-weight format is maybe to start with context-free grammars (which notation is also supported by GF). Context-free grammars that give acceptable even though over-generating results for languages like English are quick to produce.

The experience has led to the following steps for grammar development. While giving the work a quick start, this recipe increases abstraction at a later level, when it is time to to localize the grammar to different languages. If context-free notation is used, steps 1 and 2 can be merged.

1. Encode domain ontology in and abstract syntax, `Domain`.
2. Write a rough concrete syntax in English, `DomainRough`. This can be oversimplified and overgenerating.
3. Reimplement by using the resource library, and build a functor `DomainI`. This can be helped by *example-based grammar writing*, where the examples are generated from `DomainRough`.
4. Instantiate the functor `DomainI` to different languages, and test the results by generating linearizations.
5. If some rule doesn’t satisfy in some language, use the resource in a different way for that case (*compile-time transfer*).

2.2 Two strategies of multimodal fusion in DelfosNCL

This section compares two strategies of multimodal fusion of input modalities coming from different channels, and their implementation in the in-home domain dialogue system developed by the University of Seville.

Two strategies have been implemented for comparison purposes: the first solution is largely based on Johnston’s work [Johnston et al., 1997, Johnston, 1998], and involves modifying our parser to cope with

simultaneous multimodal inputs, and to include temporal constraints at unification level. The second implementation proposes an original solution to the problem, and involves combining inputs coming from different multimodal channels at dialogue level. This solution is based on an implementation of the ISU approach [Traum et al., 1999].

These two strategies have been implemented in DelfosNCL, an ISU based system, combining both speech and graphical inputs within a multimodal in-home scenario where the user interacts with the system using a microphone and a touch-screen.

2.2.1 From speech-only to multimodal interaction

Before any further considerations, some preliminary steps had to be taken in order to make the system work multimodally. The first step involved moving from a synchronous, system-driven, turn taking approach to an asynchrounous, mixed-initiative model. We faced this evolution by means of an intermediate (input pool) layer whose role is to store all inputs coming from the user at any time and make them available to the system when requested. The input pool was implemented as an independent OAA agent. The second step involved modifying the GUI interface [Quesada et al., 2000], which was originally just a floor plan representation of the house designed to configure the distribution of devices and functionalities. The new extended version of the GUI allows the user to refer to parts of the house by clicking on them with the pen. The third step was to make the speech-only input pool a multimodal input pool. This goal was achieved by allowing different kinds of inputs and storing them in a simple FIFO queue (see Figure 2.1). Namely, the multimodal input pool accepts two kinds of inputs:

- **SPEECH**, including the following fields:
  `init_time`, `end_time`, `sentence_score`, `list[word, word_score]`.
- **CLICKs**, including the icon and time fields.

For multimodal information rendering we have implemented at this stage a basic heuristic-based presentation layer which is out of the scope of this document. A global view of how the system interacts with the user is illustrated in Figure 2.2.

2.2.2 Multimodal Fusion: Two Strategies

Strategy 1

The first strategy implemented follows Johnston's proposal [Johnston et al., 1997, Johnston, 1998], by using a unification-based parser and including modality and temporal constraints at unification level. This implementation differs from Johnston's in that a higher level of flexibility is added.

The main motivation behind this strategy is that multimodality is conceived of as a single communicative act between two participants, and as such should be treated by a single grammar which is capable of accepting input coming from different modalities. As expected, this system permits that the communicative act may range from speech-only to clicks-only or hybrid inputs, and all are considered equal as far as the grammar is concerned. Obviously, as described below, this is an advantage as long as only single-task interactions and not multiple task interactions are considered. The pragmatic ambiguity which may occur in multimodal multi-tasking cannot be resolved by a single grammar. Graphically, this strategy fuses inputs at our NLU module, as illustrated in Figure 2.3.

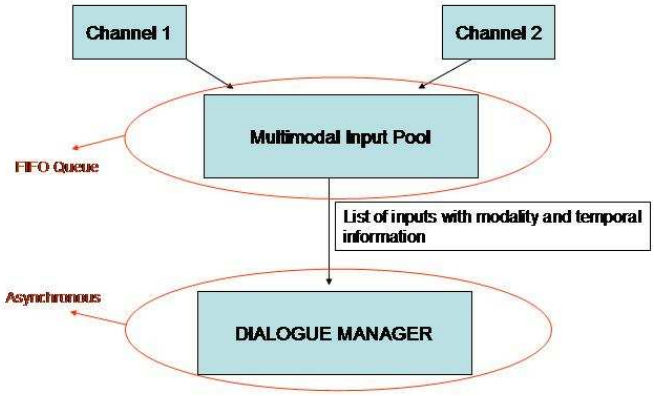

Figure 2.1: Multimodal Input Pool

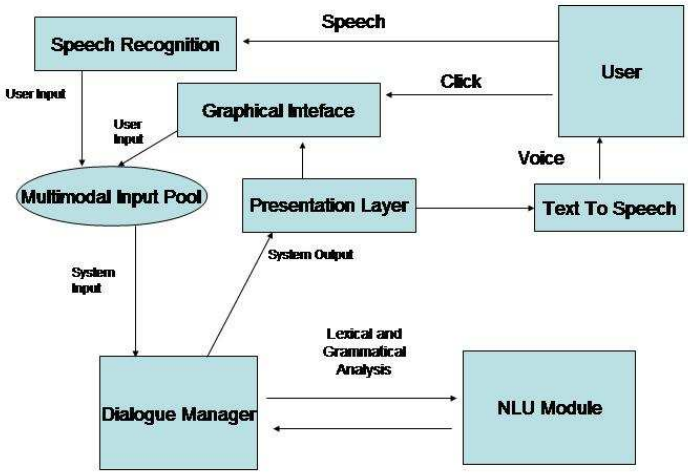

Figure 2.2: Modules overview

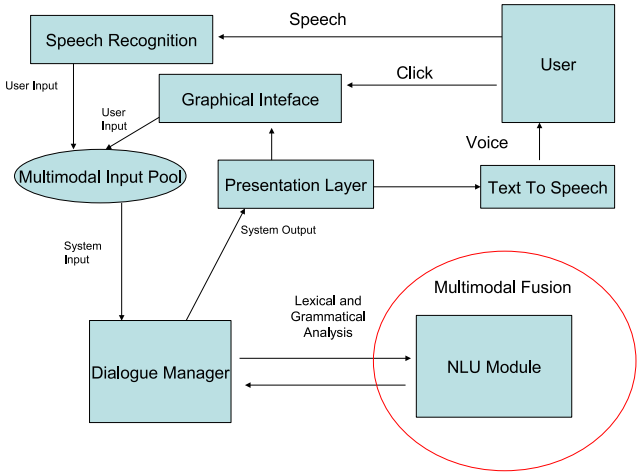

Figure 2.3: Strategy 1

In this strategy, when the parser receives an input sentence (either speech-only, click-only or mixed), it calls the lexical analyzer adding three new *ad-hoc* feature-value pairs: `MODALITY`, `TIME_INIT` and `TIME_END`. These features are then used in conjunction with a set of logical operators to define complex expressions in order to enforce modality and temporal constraints. Let's imagine a grammar rule for an input as "*switch on the light*", where light can be either specified by voice or clicked. Let's imagine as well that when using the mixed modality input (that is to say: when clicking on the light icon, the user actually clicks before uttering "*switch on*". In this case, a rule for the voice only inputs (therefore with natural command + parameter order) could be specified, and another one that only applies to mixed inputs where the inverse order is accepted (parameter + command). The unification rule will look like the following one:

```
(Rule 1 : Command -> CommandOn DeviceSpecifier)
    { @up = @self-1; }
(Rule 2 : Command -> DeviceSpecifier CommandOn)
    { @up.DeviceSpecifier =a @self-1;
      @if((@self-1.MODALITY == CLICK) && (@self-2.MODALITY == VOICE))
      @then { @if ((@self-1.TIME_INIT - @self-2.TIME_INIT <= 5) &&
                   (@self-1.TIME_INIT - @self-2.TIME_INIT <= -5))
              @then { @break();}
              @else { @up.MODALITY =a [VOICE,CLICK];
                      @if ((@self-1.TIME_INIT <= @self-2.TIME_INIT))
                      @then { @up.TIME_INIT =a @self-1.TIME_INIT;}
                      @else { @up.TIME_INIT =a @self-2.TIME_INIT;}
                      @if ((@self-1.TIME_END >= @self-2.TIME_END))
                      @then { @up.TIME_END =a @self-1.TIME_END;}
                      @else { @up.TIME_END =a @self-2.TIME_END;} } }
      @else { @break(); } }
```

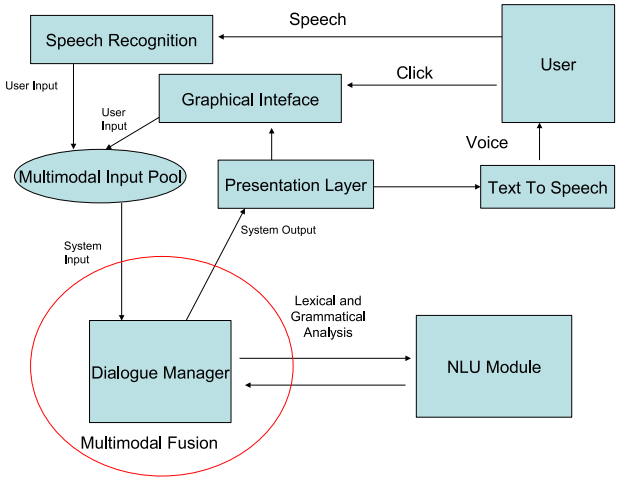

Figure 2.4: Strategy 2

Note that, in addition to the modality constraint, an overlap timeframe (5 time units) within which the inputs have to occur, has been defined. These timeframes could be configured independently (rule by rule) if the data was accurate enough. These rules describe under what conditions the right-hand symbols can unify and, if the conditions are met, how the unification has to be done.

Notice that we are not using only temporal data as subcategorization edges but actually letting the user configure the constraints case by case. However it looks like this flexibility is not always needed, so we have implemented a set of macros to be used at unification level to cover a number of cases:

1. `@assign_modality(@self-1,@self-2,@self-n)`
   Check if the modality of all the constituents is the same, otherwise, assign `MODALITY:[MIXED]` to the mother node.

2. `@assign_time_init(@self-1,@self-2,@self-n)`
   Get the lowest time init and assign it to the mother node

3. `@assign_time_end(@self-1,@self-2,@self-n)`
   Get the highest time end and assign it to the mother node

Strategy 2

The second strategy combines simultaneous inputs coming from different channels (modalities) at Dialogue Level. The idea is to check the multimodal input pool before launching the actions expectations, while holding during the *inter-modality time*. Obviously, this strategy takes for granted that each individual input can be considered as an independent Dialogue Move.

As illustrated in Figure 2.4, this strategy fuses the multimodal inputs at dialogue level. In this approach, the multimodal input pool receives and stores all inputs including information such as time and modality. The Dialogue Manager checks the input pool regularly to retrieve the corresponding input. If more than one

input is received during a certain timeframe, they are considered simultaneous or pseudo-simultaneous. In this case, further analysis is needed in order to determine whether those independent multimodal inputs are truly related or not. If the inputs timing with respect to each other is deemed to be within the plausible time range to consider them a potential multimodal combination, then additional information will be taken into account to decide whether these independent DMoves are complementary or not:

- If one is TriggeringCondition of a Dialogue Rule, and the other one is part of the expectations
- If both are expectations of an already active Dialogue Rule
- If there is no other parallel dialogue history whose active Dialogue Rules may conflict with the previously identified one

When everything indicates that the DMoves are related and complementary, they merge into a unique Information State. Otherwise, different paths may be taken depending on the situation:

- One of them may complete an already active Dialogue Rule whereas the other may trigger a new unrelated TriggeringCondition and therefore a new parallel dialogue history.
- Each of them may complete already active Dialogue Rules in parallel dialogue histories unambiguously.
- Rules in parallel dialogue histories in an ambiguous manner, in which case disambiguation subdialogues will be needed.
- They are unrelated and not compatible with any active Dialogue Rule, so two new tasks with their respective dialogue histories will be initiated.

Our approach can be described by this high-level algorithm:

- Receive unimodal input A (DMove)
- Receive unimodal input B (DMove)
- IF A and B are: complementary,
  contextually appropriate, and
  within a predefined timeframe
  THEN Create new IS from these DMoves + Dialogue History
- ELSE Store the DMove and disambiguate

This algorithm takes into account:

1. Dialogue Moves generated
2. Modality
3. Inter-input timing
4. Dialogue Move order

5. Existing Dialogue Moves
6. Existing Dialogue Histories
7. Scenario and contextual factors

Dialogue Rules may also be configured with the same logical operators mentioned within Strategy 1, since the Dialogue Manager actually uses the unification module of the parser. Similar rules to the one detailed for Strategy 1 could be configured within the Dialogue Manager. The difference is *where* these rules are applied: for Strategy 1 the scope of application is the list the symbols (terminals and not terminals) within the grammar rules, whereas the scope for Strategy 2 is the number of DTAC structures that describe the DMoves.

Although taking into account a considerable number of factors may not appear as a very appealing solution, this innovative approach enables the system to cope with "Multimodal Multitasking", which would not be possible within the implementation of Strategy 1. By Multimodal multitasking we imply the possibility of accomplishing independent unrelated tasks simultaneously, sparing continuous system disambiguation. Humans have often proven to be able and even prefer to accomplish several tasks at once, as long as they are familiar with the tools and/or environment and none of the tasks imply too heavy a cognitive load. With this approach, multimodal systems have taken a step forward towards more intelligent, flexible and collaborative systems.

2.2.3 Comparison of Strategies

Computational efficiency: The first strategy is much heavier from a computational point of view since tasks are added at unification level which represents 80% of the parsing time. On the other hand, the additional computational complexity added by the second strategy is of no consequence.

Dependency on time measures: The first strategy is highly dependent on the precision of the time data. The overlapping times fixed at unification rules assume that the `init_time` and `end_time` features are accurate, which is not always the case. The second strategy however allows for a certain degree of flexibility.

Background data: In order to define the appropriate time ranges for multimodal complementary inputs, real user data is required. The more precise this time ranges need to be, the more important it becomes to collect large amounts of data, especially considering the possibility of tuning the thresholds rule by rule.

Multimodal multitasking: The multimodal multitasking is the ability to carry independent tasks at the same time by means of different multimodal channels. The notion of task only exists at dialogue level, therefore strategy one cannot be applied if dealing with multimodal multitasking.

Inter-modality disambiguation: When dealing with more complex modalities (i.e. voice and gesture recognition) we may expect not only pairs of item-time, but full lattices coming from both channels. The mutual disambiguation could be more easily dealt with by the first strategy. The second strategy would become considerably more complex

Dialogue Act: At a theoretical level, a potential problem of the second strategy could arise from the assumption that any unimodal input generates always a Dialogue Move. Although we have been

unable to find any example or situation where this assumption is false, it could possibly be the case with more sophisticated not speech-driven systems.

Number of modalities: We believe that as the number of modalities increases, the best choice would be the second strategy, since the first strategy implies a high computational overload which would become unbearable with a higher number of modalities.

2.3 Summary

In this chapter we have described two approaches to adding integrated multimodality and multimodal fusion to unimodal dialogue systems.

The first approach is to specify multimodal input directly in GF grammars, where the modalities are realised as discontinuous constituents in the grammar. We have given a language- and domain-independent method for how to add multimodal information to a unimodal grammar, thus making the transition from a speech-only dialogue system to a multimodal one in many ways trivial. As a pedagogical example we described a multimodal version of the UGOT *Tram Demo* grammar.

The second approach for evolving from a speech-only system to a multimodal one, is by implementing an intermediate layer in DelfosNCL called a “multimodal input pool” whose role is to allow for asynchronous behaviour. The general steps taken to cope with both speech and clicking inputs have been described and two strategies to fuse multimodal entries have been explained and compared. The first strategy is to combine the multimodal inputs in the unification-based parser, and the second strategy is to combine the multimodal inputs at the dialogue level. For efficiency reasons we conclude that the second strategy suits better the needs of the DelfosNCL voice-and-click application within the in-home domain scenario.

Chapter 3

Description of the Multimodal Grammar Library

This chapter consists of a description of the unified GF/GoDiS grammar library, written in Grammatical Framework. The grammars in the library have been made multimodal by using the method described in section 2.1.1.

The grammar library connects user and system utterances specified in GF with a dialogue system using the GoDiS dialogue manager. The library is designed for making is easy to add new dialogue domains, source languages, and input and output modalities. Currently the library consists of two dialogue domains, each with two source languages and three different modalities. The two domains are the calendar application *AgendaTalk*, and the MP3 player *DJ GoDiS*.

An additional GF grammar is described in section 3.6, which has been made multimodal using the given method. The Edinburgh *Town Info* grammar is not part of the GF/GoDiS grammar library, but can be seen as a proof-of-concept of the generality of the method. Another additional grammar, the *Tram Demo* grammar, is used as a pedagogical example when introducing the method in section 2.1.1 and is therefore not described in this chapter.

3.1 The GF/GoDiS grammar library

3.1.1 The module hierarchy of the GF/GoDiS grammar library

The modules of the GF/GoDiS grammar library are divided into four main parts – ontologies, the core GoDiS grammar, the different domain grammars, and general resource modules.

Ontologies

In this grammar library we mean by the term ontology, a self-contained grammar capturing a domain which can be used in several different dialogue systems. An ontology *O* consists of at least three grammar modules – one abstract and the English and Swedish concrete modules. Apart from these there can be resource modules for simplifying grammar writing.

All ontologies are located in the directory `Ontology` in the grammar library, meaning that ontology *O*

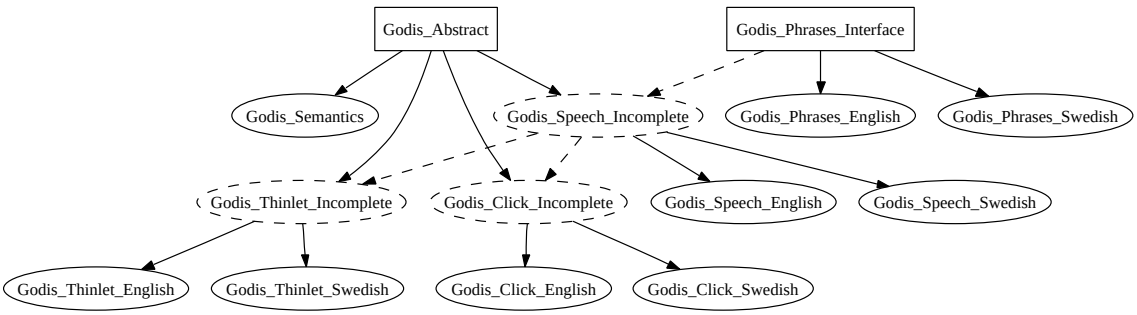

Figure 3.1: The module structure of the core GoDiS grammar

will be named `Ontology_O`. Note that this is not an existing GF module, but instead they are named `Ontology_O_M`, where $M \in \{$`Abstract`,`Semantics`,`English`,`Swedish`$\}$. The `Semantics` module consist of the GoDiS Prolog terms corresponding to the terms in the ontology.

If some ontologies can be naturally grouped together into a unified ontology, they are defined as sub-ontologies. An example is `Ontology_Music`, which consists of `Ontology_Music_Albums`, `Ontology_Music_Artists` and `Ontology_Music_Songs`.

The core GoDiS grammar modules

The grammar for the core of the GoDiS-based dialogue systems is located in the directory `Godis`, with the module structure shown in figure 3.1. The abstract syntax module is `Godis_Abstract`, and the concrete syntax for communicating with the GoDiS dialogue manager is called `Godis_Semantics`.

The concrete system and user grammars exist as a unimodal variant (`Speech`), a parallel multimodal variant for communicating via the Thinlet GUI (`Thinlet`), and an integrated multimodal variant (`Click`). For each of these there is an incomplete concrete module which is instantiated by the languages English or Swedish. Finally there are resource modules (called `Phrases`) implementing a small library of canned phrases, as an interface which is instantiated with English and Swedish. These resources are used in the concrete grammars `Speech`, `Thinlet` and `Click` – dependencies which are not shown in the figure.

This means that the core GoDiS grammar modules are the following ones:

- `Godis_Abstract`
- `Godis_Semantics`
- `Godis_`*Src*`_`*Lng*, where *Src* $\in \{$`Speech`,`Thinlet`,`Click`$\}$ and *Lng* $\in \{$`Incomplete`,`English`,`Swedish`$\}$
- `Godis_Phrases_`*Lng* where *Lng* $\in \{$`Interface`,`English`,`Swedish`$\}$

The domain grammars

All domain grammars reside in the directory `Domain`, which means that the current two domains are `Domain_Agenda` and `Domain_MP3`. In other respects they have almost the same structure as the core

grammar. What is missing are the incomplete grammar modules, and what is added is an abstract module for the integrated modality module `Domain_`*Dom*`_Click`.

The grammar modules are thus the following, where *Dom* $\in$ {`Agenda`, `MP3`}:

- `Domain_`*Dom*`_Abstract`
- `Domain_`*Dom*`_Semantics`
- `Domain_`*Dom*`_Click_Abstract`
- `Domain_`*Dom*`_`*Src*`_`*Lng*, where *Src* $\in$ {`Speech`, `Thinlet`, `Click`} and *Lng* $\in$ {`English`, `Swedish`}
- `Domain_`*Dom*`_Phrases_`*Lng* where *Lng* $\in$ {`Interface`, `English`, `Swedish`}

General resource modules

Some resource modules are very general, and do not fit in any of the three previous categories, instead they are placed in the directory `Resource`. These resources contain operations for simplifying grammar writing in formal languages, such as Prolog, XML and Thinlet GUI. The corresponding modules are named `Resource_Prolog`, `Resource_XML` and `Resource_Thinlet`.

3.1.2 Translating between user languages and GoDiS dialogue moves

The GoDiS dialogue manager does not operate on utterances or parse trees or some other kind of syntactic representation. Instead the basic input and output items are dialogue moves, which can be seen as semantic representations of the minimal informative units in a conversation [Larsson, 2002]. An utterance by the user or the system is represented as a sequence of dialogue moves.

The role of GF in this context is as a translator between dialogue moves and linguistic utterances, or even representations in other modalities. The dialogue grammars in our library all share the same structure, as explained in section 3.1.1. There is one single abstract syntax capturing all information that can be shared between the dialogue participants. For each different kind of uni-/multi-modality and each different language, there is a concrete syntax module.

Finally there is one single concrete module describing the GoDiS dialogue moves. As will be seen later in section 3.3, the structure of the abstract syntax is similar to the hierarchical structure of the GoDiS dialogue moves, which means that the definition of the GoDiS concrete syntax module is quite straightforward.

In the dialogue system the translation takes place as follows. A user utterance is parsed to an abstract syntax term which is then linearized to a sequence of GoDiS dialogue moves. The dialogue moves are interpreted by the dialogue application which generates new dialogue moves. These are in turn parsed and linearized by the GF interpreter to system utterances.

3.1.3 Resources used in the grammar library

We have used two kinds of resource modules in the GF/GoDiS grammar library – linguistic and non-linguistic resources.

Linguistic resources

The linguistic resource modules consist of language-specific parameters, and definitions of common phrases and utterances. The common phrases and utterances are defined as interface modules, and instantiated for each surface language. This is used in the core GoDiS dialogue system grammars, where each different modality consists of one incomplete concrete syntax which only makes use of language-specific phrases and utterances from an interface module. The different surface level languages then consist of simple instantiations of interface modules.

This structure makes it simple to add a new language. The only thing we have to do is to create a new instance module for the common phrases – the core GoDiS grammar does not have to be changed at all.

Non-linguistic resources

The non-linguistic resources consist of macros for simplifying grammar writing in formal languages, such as Prolog and XML.

The Prolog resource module makes it simple to translate GF trees into compound terms in Prolog syntax. This is used in the semantics modules of the core GoDiS grammar, the ontologies and the domain specific grammars.

The XML resource module contains macros for linearizing GF trees as XML expressions. This in turn is used in the Thinlet resource module with which we can create GUI objects described in Thinlet XML syntax.

There is also a resource module which defines the linearization types that are used in the GF/GoDiS grammars, and operations for simple creation of terms of these linearization types. The different participants of the dialogue system are defined as parameters, together with helpful operations. Some of these parameters, operations and linearization types are described later in section 3.4.

3.2 Grammars for describing ontologies

In this grammar library we mean by the term ontology, a self-contained grammar capturing a domain which can be used in several different dialogue systems. We divide the ontology grammars into *databases* and *linguistic ontologies*. A database mainly consists of a listing of elements of one or a few categories, such as a listing of all known artists or possible event types that can be stored in an agenda. A linguistic ontology on the other hand consists of a more elaborate grammar, describing a linguistic concept such as numbers or time descriptions.

3.2.1 Databases

Music – Artists, Albums, Songs

The music ontology is divided into three sub-ontologies: artists, albums and songs. Each ontology grammar defines a GF category (`Artist`, `Album` and `Song`, respectively) together with its entities.

Radio and TV stations

These ontologies contain names of radio and TV stations.

Events and locations

These ontologies contain possible kinds of events that can be stored in an agenda, and possible locations where the events can take place.

3.2.2 Linguistic ontologies

Numbers

This ontology defines the numbers from 0 to 99, in both cardinal and ordinal notation.

Time descriptions

This ontology defines hours and minutes, and how to combine these into a linguistically correct time expressions.

Date descriptions

Finally, the ontology of date descriptions defines weekdays and relative date expressions (e.g. today, tomorrow).

3.3 The GF/GoDiS dialogue move grammar

In this section we present the type hierarchy of GoDiS and explain how this has influenced the abstract syntax of the GF grammar. The version of GoDiS that we describe is *action-oriented-dialogue* [Larsson, 2002, chapter 5].

3.3.1 The type hierarchy in GoDiS

In this section we give an overview of the hierarchy of types in GoDiS. For a more thorough description, see Larsson [2002].

Domain-dependent types

A domain consists of a number of *individuals*, which are grouped into *sorts*:

- $n_1, \ldots, n_N$: Ind
- $s_1, \ldots, s_M$: Sort

For forming propositions and questions, there are a number of basic *predicates*, which can be either atomic (Pred0) or take one argument (Pred1):[1]

- $p_1, \ldots, p_N$: Pred0 or Pred1

To form requests for actions, there is a number of basic *actions*:

- $a_1, \ldots, a_N$: Action

The domain also consists of the *dialogue participants*, which in our case only are the *user* and the *system*:

- user, system : Participant

Finally, there are some *reasons*, which are used when reporting success or failure:

- $r_1, \ldots, r_N$: Reason

The rest of the types and objects are domain-independent, and are described in the rest of this section.

Dialogue moves

A *dialogue move* is the basic entity that the update rules in a GoDiS dialogue system works on. We can distinguish between six different kinds of dialogue moves. *Single dialogue moves* consist of greeting and quitting, in the beginning and the end of a conversation:

- greet, quit : SingleMove

An action can be *requested*, and a question can be *asked*:

- request(a) : Request [a : Action]
- ask(q) : Ask [q : Question]

There are two kinds of *answers*, propositions and short answers, which are semantically underspecified propositions:

- answer(p) : Answer [p : Proposition or p : ShortAnswer]

The *report* dialogue move handles reporting of success and failure of actions, sometimes giving a reason for why the action failed:

- report(a, failed(r)) : Report [a : Action, r : Reason]
- report(a, done), confirm(a) : Report [a : Action]

Interactive Communications Management (ICM), is used as a general term for coordination of the common ground. The different kinds of ICM dialogue moves are described later in the end of this section.

[1] We don't need higher arity than 1, since GoDiS uses a reduced semantic representation, where complex predicates are replaced by (a number of) 1-place predicates.

Propositions

GoDiS uses a reduced semantic representation, where complex propositions with conjunctions and *n*-ary predicates are represented as sets of 1-ary predicates applied to constant individuals:

- p : Proposition [p : Pred0]
- $p(n)$: Proposition [p : Pred1, n : Ind]

GoDiS uses a rudimentary system of domain-dependent semantic sortal categories, for distinguishing meaningful propositions from meaningless ones. What this amounts to is the restriction that $p(n)$ is a meaningful proposition only if p and n are associated with the same sort s. In the GF grammar library we encode the association to a sort as a type dependency, i.e. Pred1(s) and Ind(s) are GF categories only if s : Sort.
Disjunction is not present, and conjunction is not needed, so the only logical connective we need is negation:

- not(p) : Proposition [p : Proposition]

Finally, there are a number of ways of talking about questions, actions and propositions on a meta-level:

- issue(q), fail(q), fail(q, r) : Proposition [q : Question, r : Reason]
- action(a), done(a) : Proposition [a : Action]
- und(d, p) : Proposition [d : Participant, p : Proposition]

Note that these propositions can be viewed as applications of predicates, but they need special consideration, since the arguments are not individual constants. Furthermore, they are only used in special circumstances – issue(q) and action(a) are only used in questions and ICM dialogue moves; und(d, p) is only used in ICM dialogue moves; whereas fail(q), fail(q, r) and done(a) are only used as success reports.[2] This means that they are not categorized as propositions in the GF grammar library, but as special cases for wh-questions, ICM and reports.

Short answers

Short answers are semantically underspecified propositions, and consist of yes/no-answers, individual constants, or negations of individual constants:

- yes, no : ShortAnswer
- not($s(n)$), $s(n)$: ShortAnswer [s : Sort, n : Ind]

[2] Internally in the GoDiS information state they can function as propositions, but to an external observer they are only used in these circumstances.

Questions

There are three kinds of questions – y/n-questions, alternative questions and wh-questions. They can be seen as subtypes of the type for questions:

- q : Question [q : YNQ or q : AltQ or q : WhQ]

Y/n-questions are formed from a proposition:

- ?p : YNQ [p : Proposition]

Alternative questions are sets of y/n-questions:

- $\{q_1, \ldots, q_n\}$: AltQ [q_1 : YNQ, $\ldots$, q_n : YNQ]

Wh-questions are lambda-abstractions of propositions (written with a question mark instead of a lambda), but since lambda-abstracting a negation is linguistically difficult we exclude negated propositions. This leaves us with three forms of wh-questions:

- ?$x.p(x)$: WhQ [x : Var, p : Pred1]
- ?x.issue(x), ?x.action(x) : WhQ [x : Var]

Var is the type of variables; x, y, z, $\ldots$: Var.

Interactive Communications Management (ICM)

Larsson [2002] uses Interactive Communication Management (ICM) as a general term for coordination of the common ground. ICM dialogue moves are explicit signals enabling coordination of updates to the common ground, such as keeping track of topics currently under discussion, subactivities, sequencing and turn taking.
There are two kinds of ICM dialogue move patterns in GoDiS. The main pattern deals with feedback and grounding:

- (icm : l * p), (icm : l * p : *args*) : ICM [l : Level, p : Polarity]

There are five *action levels* – contact, perception, semantic understanding, pragmatic understanding, and acceptance/reaction. These are abbreviated con, pre, sem, und and acc, respectively. There are three *polarities* – positive, negative and interrogative. These are abbreviated pos, neg and int, respectively. Some feedback moves also require *arguments*, which depending on the action level and the polarity can be either a question, a proposition, an action or a string. The String is yet another GoDiS type, consisting of all possible surface-level strings.
The second pattern for ICM dialogue moves is used for ICM other than feedback:

- (icm : *type*), (icm : *type* : *args*) : ICM

The *type* can be any of reraise, loadplan, accomodate and reaccomodate. Some of these can also take optional *arguments*.

3.3.2 Representing GoDiS types in GF abstract syntax

In this section we describe how we have translated the GoDiS type hierarchy into GF. This central GF/GoDiS grammar consists of the abstract module `Godis_Abstract`.

The obvious idea is to represent the GoDiS types as GF categories, and the GoDiS objects as GF constants and functions. However, in some places we have divided a type into several categories. There are also some other minor differences, due to some simplifying assumptions we have made on the grammar.

Domain-dependent categories

Sorts, individuals, 0- and 1-place predicates, actions and reasons are defined as basic categories in the central GF/GoDiS grammar. However, all these categories are uninhabited since their elements will be defined in the domain grammar.

```
cat Sort;
    Ind Sort;
    Pred0 Sort;
    Pred1 Sort;
    Action;
    Reason;
```

Since individuals and predicates are associated with a certain sort, these categories depend on the category of sorts. This is an example of dependent types, which we use to give restrictions for e.g. how to combine a predicate with an individual.

Note that we have excluded dialogue participants, since they are already fixed in the systems we are focussing on, and since they are anyway only used in a limited number of dialogue moves. The participants are instead hard-coded in the grammars.

A central goal of the grammar library is that (almost) every sort, together with its associated individuals, should be defined in an ontology. This means that defining the sorts and individuals of a dialogue domain merely consists of enumerating the ontologies that are used in the domain. This is done by inventing a name for the sort, and giving a coercion function from the ontology category to an individual. E.g. the sort of numbers can be defined by giving the following two functions:

```
fun NumberSort : Sort;
    numberInd  : Number -> Ind NumberSort;
```

Dialogue moves

We start by defining the basic category of dialogue moves:

```
cat Move;
```

The different kinds of dialogue moves are also basic GF categories, which are given together with coercion functions to the category of moves:

```
cat SingleMove;       fun singleMove  : SingleMove -> Move;
    YNAnswer;             yesnoMove   : YNAnswer -> Move;
    Request;              requestMove : Request -> Move;
    Ask;                  askMove     : Ask -> Move;
    Answer Sort;          answerMove  : (s:Sort) -> Answer s -> Move;
    Report;               reportMove  : Report -> Move;
    ICM;                  icmMove     : ICM -> Move;
```

Note that the answers are divided into two categories, since y/n-answers cannot be associated with a sort. This also means that y/n-answers are not considered as short answers in the GF grammars.

The *single moves* greet and quit are GF constants, as are the *y/n-answers* yes and no:

```
fun greetMove, quitMove : SingleMove;
    yesAnswer, noAnswer : YNAnswer;
```

An action can be *requested*, and a question can be *asked*:

```
fun requestAction : Action -> Request;
    askQuestion   : Question -> Ask;
```

An *answer* can be a short answer or a proposition. But since y/n-answers are already treated, the type of short answers is not implemented as a GF category. Short answers are thus elliptic individuals, possibly negated. Furthermore, as explained below, only positive propositions are considered, meaning that there is a possibility of a negated propositional answer:

```
fun indAnswer     : (s:Sort) -> Ind s -> Answer s;
    notIndAnswer  : (s:Sort) -> Ind s -> Answer s;
    propAnswer    : (s:Sort) -> Proposition s -> Answer s;
    notPropAnswer : (s:Sort) -> Proposition s -> Answer s;
```

An action can be *reported* as a success or a failure. However, we also consider failure of finding an answer to a question as a report, instead of an answer. This is partly because we do not consider fail(q) or fail(q,r) as propositions, but also because failures are often uttered in a different way than are traditional answers:

```
fun confirmActionReport        : Action             -> Report;
    failedActionReport         : Action   -> Reason -> Report;
    failedQuestionReasonReport : Question -> Reason -> Report;
    failedQuestionReport       : Question           -> Report;
```

Finally, *ICM* dialogue moves are treated later in this section.

Utterances

An utterance in GoDiS consist of a sequence of dialogue moves. Thus we define the category of utterances, and lists of dialogue moves; together with a function for forming utterances from a list of moves:

```
cat Utterance;        fun utterance : [Move] -> Utterance;
    [Move]{1};
```

Note that in GF, `[Move]` is syntactic sugar for the category `ListMove`; and by declaring `[Move]{1}`, the following two functions are silently defined:

```
fun BaseMove : Move -> [Move];
    ConsMove : Move -> [Move] -> [Move];
```

Although many dialogue moves are independent of the dialogue participant, there are some dialogue moves (such as reports) that can only be uttered by the system, and other moves (such as requests) that can only be uttered by the user. This information is used in parsing, and is encoded in the concrete syntax. For this purpose, we define the categories of user and system utterances, together with forming functions:

```
cat UserUtterance;    fun userUtterance   : Utterance -> UserUtterance;
    SystemUtterance;      systemUtterance : Utterance -> SystemUtterance;
```

Propositions

Since both predicates and individuals depend on sorts, propositions will also be associated with sorts:

```
cat Proposition Sort;
```

None of the meta-level propositions are seen as propositions in the grammar library. Furthermore, we choose to also exclude negated propositions. Some reasons for this are *i*) that we want to minimize the recursiveness in the grammar for efficiency reasons, *ii*) that negation can be difficult or complicated to represent syntactically correct in its generality, and *iii*) that negated propositions are meaningless when viewed as y/n-questions. Thus, propositions can only be formed by applications of 0- and 1-place predicates:

```
fun pred0prop : (s:Sort) -> Pred0 s          -> Proposition s;
    pred1prop : (s:Sort) -> Pred1 s -> Ind s -> Proposition s;
```

Questions

The category of questions is subdivided into y/n-questions, alternative questions and wh-questions. These are declared as categories, together with coercion functions:

```
cat Question;
    YNQ;           fun ynqQuestion : YNQ  -> Question;
    AltQ;              altQuestion : AltQ -> Question;
    WhQ;               whqQuestion : WhQ  -> Question;
```

Y/n-questions are formed from propositions, and from the meta-level propositions issue(q) and action(a):

```
fun propYNQ   : (s:Sort) -> Proposition s -> YNQ;
    issueYNQ  : Question -> YNQ;
    actionYNQ : Action   -> YNQ;
```

Alternative questions are formed from sequences of y/n-questions, which has to be defined as a category itself:[3]

```
cat [YNQ]{2};
fun altQ : [YNQ] -> AltQ;
```

Wh-questions can finally be formed from 1-place predicates:

```
fun predWhQ : (s:Sort) -> Pred1 s -> WhQ
```

There are also the two special wh-questions ?x.issue(x) and ?x.action(x):

```
fun issueWhQ, actionWhQ : WhQ
```

Interactive Communications Management (ICM)

The different ICM dialogue moves are defined by enumerating them in the GF grammar. The reason for not having some more general functions dealing with ICM is partly because different action levels and polarities can take different arguments, and partly because we want to have different surface forms for different ICM's.

There are two ICM's for acceptance, negative and positive; but since the negative version can take an optional question or proposition argument, we get in total four GF functions:

```
fun accNegICM     : ICM;
    accNegQueICM  : Question -> ICM;
    accNegPropICM : (s:Sort) -> Proposition s -> ICM;
    accPosICM     : ICM;
```

There is only one ICM for contact, and that is negative:

```
fun conNegICM : ICM;
```

There are three ICM's for perception – negative, positive and interrogative; where the positive version takes a string comprising the utterance as the system heard it:

```
fun perNegICM    : ICM;
    perPosStrICM : String -> ICM;
    perIntICM    : ICM;
```

[3]The argument `{2}` says that a list of y/n-questions has to have at least two elements. This is accomplished by giving the `BaseYNQ` function an arity of 2.

There are in total seven ICM for understanding – mostly because the positive version can take both positive and negative propositions, including issues:

```
fun undNegICM        : ICM;
    undPosPropICM    : (s:Sort) -> Proposition s -> ICM;
    undPosNotPropICM : (s:Sort) -> Proposition s -> ICM;
    undPosQueICM     : Question -> ICM;
    undPosNotQueICM  : Question -> ICM;
    undIntPropICM    : (s:Sort) -> Proposition s -> ICM;
    undIntQueICM     : Question -> ICM;
```

Finally there are eight non-feedback ICM functions, since reraising and accomodation can take questions or actions as arguments:

```
fun reraiseICM         : ICM;
    reraiseQueICM      : Question -> ICM;
    reraiseActICM      : Action   -> ICM;
    loadplanICM        : ICM;
    accomodateICM      : ICM;
    accomodateQueICM   : Question -> ICM;
    reaccomodateICM    : ICM;
    reaccomodateQueICM : Question -> ICM;
```

3.4 Concrete syntaxes for the central GF/GoDiS grammar

3.4.1 Prolog syntax for connecting to GoDiS – Semantics

The concrete grammar module `Godis_Semantics` transforms the abstract GF syntax terms into Prolog readable terms suitable for input to GoDiS. The translation is quite straightforward, since the structure of the abstract grammar reflects the GoDiS type hierarchy. There are resource modules `Resource_Prolog` and `Godis_Semantics_Resource` for creating GoDiS terms in Prolog syntax, with operations for creating simple and compound terms, lists, operator applications, and other special Prolog and GoDiS constructions.

All categories have the same linearization type PStr, which is defined in the Prolog resource module:

```
lincat Move, ... = PStr;
```

As an example, the linearizations for the four kinds of answers are defined like this:[4]

```
indAnswer    sort ind = pp1 "answer" (pp1 sort.pl ind);
notIndAnswer sort ind = pp1 "answer" (pp1 "not" (pp1 sort.pl ind));
propAnswer     _ prop = pp1 "answer" prop;
notPropAnswer  _ prop = pp1 "answer" (pp1 "not" prop);
```

[4]The first argument of `propAnswer` and `notPropAnswer` is the sort *s*, which is used in the dependent type `Proposition` *s*, but is not used in the surface form, hence the underscore.

The operations `pp0` and `pp1` (and `pp2` and `pp3...`) are defined in the Prolog resource module and create Prolog terms taking zero or one (or two or three...) arguments:

```
oper pp0 : Str -> PStr = \f -> {pl = f};
     pp1 : Str -> PStr -> PStr = f,x -> {pl = f ++ "(" ++ x.pl ++ ")"};
```

The PStr linearization type consists of only one string, under the label `pl`:

```
oper PStr : Type = {pl : Str};
```

Provided suitable linearization definitions for numbers, the following is the result of linearization in GF:[5]

```
> l (notIndAnswer NumberSort (numberInd n3)
answer ( not ( number ( 3 ) ) )
```

3.4.2 Natural language utterances – English and Swedish

We have chosen to have a very flat structure in the grammars for English and Swedish. By this we mean that there is no intricate grammatical structure inherent in the linearizations – instead they consist of canned phrases. The main reason for this is that the GoDiS dialogue system is based on sequences of dialogue moves, which are often mapped to single short phrases in an utterance. Thus, a GoDiS dialogue system has itself a flat linguistic structure to its semantics.

The only differences between the English and Swedish grammars are the canned phrases – they have the same linearization types, and the phrases have the same structure. Because of this the grammars are implemented as a single *incomplete concrete* grammar module:[6]

```
incomplete concrete Godis_Speech_Incomplete of Godis_Abstract =
    PredefCnc ** open Godis_Phrases_Interface in ...
```

The *interface* module `Godis_Phrases_Interface` consists of declarations of the canned phrases. For all languages, the declarations in the interface are instantiated in the *instance* modules `Godis_Phrases_English` and `Godis_Phrases_Swedish`.

To give the final English concrete GF/GoDiS grammar, the incomplete module `Godis_Incomplete` is completed by the respective instance modules:

```
concrete Godis_Speech_English of Godis_Abstract = Godis_Speech_Incomplete
    with (Godis_Phrases_Interface = Godis_Phrases_English);
```

The Swedish concrete grammar is created analogously.

[5] Recall that in Prolog there must not be any whitespace between a functor and its opening parenthesis, but this is handled in the tokenization phase.

[6] The predefined GF module `PredefCnc` defines the category of *strings*, which is used in the ICM move for positive perception.

Common phrases

Phrases and substrings that might occur in different places in a grammar are preferrably put in a resource module. We go one step further and use the same phrases for both English and Swedish.[7] The canned phrases can then be defined in an *interface* module:

```
interface Godis_Phrases_Interface = {
  oper yesS       : Str;
       noS        : Str;
       no_waitS   : Str;
       i_want_toS : Str;
       i_wonderS  : Str;
       ...
```

The interface can now be instantiated by a specific language in an *instance* module:

```
instance Godis_Phrases_English of Godis_Phrases_Interface = {
  oper yesS       = variants{"yes"; "yup"; ["that's correct"]};
       noS        = variants{"no"; "nope"};
       no_waitS   = optStr noS ++ "wait";
       i_want_toS = "I" ++ variants{["want"]; ["would like"]} ++ "to";
       i_wonderS  = variants{["I wonder"];
                             i_want_toS ++ variants{"ask";"know"}}
                    ++ optStr (variants{"if";"whether"});
       ...
```

Note that we can reuse phrases when defining new phrases. The operation `optStr` makes its string argument optional:

```
oper optStr : Str -> Str = \s -> variants{[]; s};
```

Dialogue participants

In the dialogue systems we are considering, some dialogue moves are system specific and some are user specific, and there are also dialogue moves that can be uttered by any dialogue participant. For this purpose we define dialogue participant as a GF parameter:

```
param Participant = System | User | Both;
```

Now, most of the categories in the GF/GoDiS grammar will have linearization types with an inherited argument saying which participant is allowed to use a given term. This is accomplished by extending the linearization types with the record `Who`, and a helper function for creating `Who` records:

```
oper Who : Type = {who : Participant};
     who : Participant -> Who = \p -> {who = p};
```

[7] Note that this approach might be more difficult if the languages are not similar.

Dialogue moves

First we define the basic linearization type of string records, together with a function for forming string records:

```
oper SS : Type = {s : Str};
     ss : Str -> SS = \str -> {s = str};
```

The linearization type for a dialogue move consists of a string record extended with a dialogue participant:

```
oper SMove : Type = SS ** Who;
lincat Move, SingleMove, Request, Ask,
       YNAnswer, Answer, Report, ICM   = SMove;
```

We also define some helper operations for creating dialogue moves. The main operation takes a participant and a string as arguments, and the other ones fixes the participant:

```
oper sMove   : Who -> Str -> SMove = \w,s -> w ** ss s;
     sBoth   :        Str -> SMove = sMove (who Both);
     sUser   :        Str -> SMove = sMove (who User);
     sSystem :        Str -> SMove = sMove (who System);
```

Now we can say that action requests can only be uttered by the user, and that reports are system-specific:

```
lin requestAction act = sUser (optStr i_want_toS ++ act.s ++ pleaseS);
    confirmActionReport act = sSystem (act.sDecl);
```

Note the two constituents of an action – one for the user's request ("play some music"), and another for the system's confirmation ("Okay, playing some music").

Ambiguity and system utterances

Both user and system utterances are described in the same concrete grammar module, which yields a small conflict when there are several alternative surface forms for the same utterance – which of the possibilities is the preferred one for system utterances. An example is the ICM move for negative perception, which could e.g. be uttered as "I'm sorry, I didn't get that", "what did you say" or simply "what". This can be encoded in GF with the `variants{...}` construction:

```
lin perNegICM = sBoth (variants{ ["I'm sorry, I didn't get that"];
                                 ["what did you say"]; ["what"] });
```

We choose to take the first of the possible variants as the default system utterance, meaning that whereas the user can say any of the three variants for the same dialogue move, the system will always say "I'm sorry, I didn't get that". This means that the system and the user speak the same language – i.e. that the system can recognize its own utterances. This is good since the user tends to speak in the same way as the system does (see, for example, Pickering and Garrod, 2004).

Utterances as sequences of dialogue moves

An utterance is made from a sequence of moves, which in turn is created by linearizing the functions `BaseMove` and `ConsMove`:

```
lincat [Move], Utterance = SMove;
lin BaseMove m    = m;
    ConsMove m ms = combineWho m ms ** ss (m.s ++ ms.s);
    utterance  ms = ms;
```

When combining two moves we check that both moves are associated with the same participant:[8]

```
oper combineWho : Who -> Who -> Who = \w1,w2 ->
                  who (case <w1.who,w2.who> of
                  { <p,Both>        => p;
                    <Both,p>        => p;
                    <System,System> => System;
                    <User,User>     => User;
                    _               => variants{} });
```

User and system utterances are those utterances which can be uttered by the user and the system respectively. Then we do not need to include a participant in their linearization types:

```
lincat UserUtterance, SystemUtterance = SS;
lin systemUtterance utt = checkSystem utt;
    userUtterance   utt = checkUser   utt;
```

In these linearizations we make use of operations for ensuring that a dialogue move can be uttered by a given participant:

```
oper checkSystem : SMove -> SS = \m -> case m.who of
                   { System | Both => m; _ => variants{} };
     checkUser   : SMove -> SS = \m -> case m.who of
                   { User | Both => m; _ => variants{} };
```

Propositions

The surface form of propositions consists of a string and a dialogue participant:

```
lincat Proposition = SMove;
```

Propositions are formed from predicates, where a 0-place predicate is a proposition of its own, and a 1-place predicate is applied to an individual:

```
lin pred0prop _ pred     = pred;
    pred1prop _ pred ind = apply pred ind;
```

The linearization operation `apply` is defined below when we describe 1-place predicates.

[8] Note that due to record subtyping, the `combineWho` operation can be applied to any linearization type which is an extension of `Who`, such as `SMove` as in the definition of `ConsMove`.

Questions

A question can be used in two different contexts – either directly as a question (“Which artist do you mean?”), or indirectly by talking about a question (“We’re talking about which artist you mean”). Also, in a specific dialogue domain, different questions can be asked by the system (“What song do you want to listen to?”) and the user (“Which song is playing now?”). This suggests that the linearization type for questions consists of two strings and the dialogue participant:

```
oper SQuestion = SQue ** Who;
     SQue      = {sQue : Str; sInd : Str};
lincat Question, YNQ, [YNQ], AltQ, WhQ : SQuestion;
```

Y/n-questions are formed from propositions. The dialogue participant is the opposite of the participant of the proposition – if one participant can propose a proposition, then the other participant can ask if the proposition is true:

```
lin propYNQ _ prop = switchWho prop **
                     prefixQue ["is it true that"] (prop.s);
```

Switching participants is defined as an operation:

```
oper switchWho : Who -> Who = \w -> who (case w.who of
                 { System => User; User => System; Both => Both });
```

The operation `prefixQue` prefixes an indirect question to form a direct question:

```
oper prefixQue : Str -> Str -> SQue = \prefix,indir ->
                 {sQue = prefix ++ indir; sInd = indir};
```

Y/n-questions can also be formed from questions and actions, and they can only be asked by the system:

```
lin  issueYNQ  que = {who = System} **
                     prefixQue you_wantS (["ask about"] ++ que.sInd);
     actionYNQ act = {who = System} **
                     prefixQue you_wantS (act.s);
oper you_wantS     = ["do you want to"];
```

Alternative questions are formed from lists of questions:

```
lin altQ ynqs = ynqs;
```

All questions occuring in a list of y/n-questions have to incorporate the same participant, which is done by the `combineWho` operation:

```
lin BaseYNQ ynq ynq' = combineWho ynq ynq' **
                       prefixQue you_wantS (ynq.sInd ++ ["or"] ++ ynq'.sInd);
    ConsYNQ ynq ynqs = combineWho ynq ynqs **
                       prefixQue you_wantS (ynq.sInd ++ [","] ++ ynqs.sInd);
```

Since all alternative questions that occur in our dialogue system consist of issue/action-questions, we can use `prefixQue you_wantS` to get aggregation; i.e. so that the system can ask "do you want to A, B or C" instead of asking "do you want to A, do you want to B or do you want to C".[9]

Wh-questions are formed from 1-place predicates, with switched participants. This means that if it is the user who asks a question about e.g. the current song, then it is the system who will give the answer:

```
predWhQ _ pred = switchWho pred ** pred;
```

There are also the two special system-only wh-questions about actions and issues:

```
actionWhQ = {who = System;
             sQue = ["what can I do for you"];
             sInd = ["how I can help you"]};
issueWhQ  = {who = System;
             sQue = ["do you need some information"];
             sInd = ["what kind of information you need"]};
```

ICM

Most ICM moves can only be uttered by the system, except negative and positive acceptance and negative perception which can be uttered by both participants:

```
lin accNegICM = sBoth i_dont_knowS;
    accPosICM = sBoth okayS;
    perNegICM = sBoth whatS;
```

Some other examples of ICM moves are:

```
lin accNegQueICM    q = sSystem (["I can't answer questions about"] ++ q.sInd);
    undIntPropICM _ p = sSystem (p.s ++ [", is that correct ?"]);
    reraiseActICM   a = sSystem (["Returning to"] ++ a.s);
```

Domain-specific categories

Sorts, individuals and reasons are simply strings:

```
lincat Sort, Ind, Reason = SS;
```

Actions are used either in user requests ("play some music") or in system confirmations ("Okay, playing some music"):

```
oper SAction : Type = SS ** {sDecl : Str};
lincat Action = SAction;
```

[9] A better way of solving aggregation is to use GF transfer rules, but this is an experimental feature still under development, so we have decided not to incorporate this in the current version of the grammar library.

0-place predicates are only used as propositions, meaning that they have the same linearization type:

```
lincat Pred0 = SMove;
```

1-place predicates are the most complicated – they can be used as answers and questions. Furthermore, when they are used as answers they are applied to an individual, which depending on the predicate can occur in different places in the phrase. E.g. “it is *Madonna* who has made the song”, or “you’re listening to *Madonna*”. Therefore we use a string with a hole as the answer linearization:

```
oper SPred : Type = Who ** SHole ** SQue;
lincat Pred1 = SPred;
```

A string with a hole is in reality implemented as two discontinuous strings:

```
oper SHole : Type = {s1 : Str; s2 : Str};
```

Now, a string with a hole can be applied to an ordinary string to form a string, which is used when forming a predicate proposition from a predicate and an individual:

```
oper apply : Who ** SHole -> SS -> SMove
           = \p,n -> sMove p (p.s1 ++ n.s ++ p.s2);
```

3.4.3 Parallel multimodality – Thinlet GUI XML-format

System output can be not only speech, but also presentations in a graphical user interface. We have chosen to use the Thinlet GUI toolkit[10] for alternative representations. Thinlet GUI components, widgets, are described in XML format and are rendered as java AWT components. Most standard AWT components, e.g. buttons, lists and text fields, are supported. For event handling, any public Java method accessible for the Thinlet component can be called. The widgets can be referenced by using the `name` attribute.

Graphical representation of dialogue moves

We represent a list of dialogue moves as a panel widget, where widgets representing the individual moves are added in a top-down fashion; i.e. the first move is at the top of the panel, and the last move is at the bottom:

```
<panel name="godis-output" columns="1">
    Move_1-Widget
    ...
    Move_n-Widget
</panel>
```

[10]The Thinlet GUI toolkit is described at and can be downloaded from `http://www.thinlet.com/`.

For dialogue management purposes, it is important to distinguish between interactive and non-interactive widgets. Non-interactive widgets are represented as label widgets, where the text on the label is the natural language representation of the dialogue move. Interactive widgets are represented as panels containing clickable buttons which are associated to a specific input dialogue move. When clicking a button, the dialogue move is written to the dialogue manager.

Currently only questions (except for wh-questions) are represented as interactive widgets, while other kinds of moves are non-interactive. A simple non-interactive move is the quit move:

```
<panel columns="1">
    <label text="Goodbye."/>
</panel>
```

An answer is represented as a panel containing the move's natural language representation as labels. For instance, the dialogue move "answer(songs_by_artist(like_a_prayer))" with the English representation "Like a prayer" is translated to the following widget:

```
<panel columns="1">
   <label text="Like a prayer"/>
</panel>
```

A yes/no-question is represented as a panel containing a label and two buttons. The text on the label is the natural language representation of the yes/no-question, the buttons correspond to the yes and no answer, respectively:

```
<panel columns="1">
    <label text="Do you want to pause?"/>
    <button name="answer(yes)" text="Yes" action="input(this.name)"/>
    <button name="answer(no)"  text="No"  action="input(this.name)"/>
</panel>
```

Also the ICM move "icm:und*int:usr*artist(madonna)" with the English representation "Madonna, is that correct?" is translated to:

```
<panel columns="1">
    <label text="Madonna, is that correct?"/>
    <button name="answer(yes)" text="Yes" action="input(this.name)"/>
    <button name="answer(no)"  text="No"  action="input(this.name)"/>
</panel>
```

Alternative questions are represented as a panel containing one button for each alternative plus a cancel button, used to reject the whole alternative question. The dialogue move

ask({?action(handle_player), ?action(handle_playlist), ?action(handle_stations)})

with the English representation "Do you want to control the player, manage playlists or listen to radio?" is represented as:

```
<panel>
    <button name="answer(action(handle_player))"
            text="Control the player" action="input(this.name)"/>
    <button name="answer(action(handle_playlist))"
            text="Manage playlists" action="input(this.name)"/>
    <button name="answer(action(handle_stations))"
            text="Listen to radio" action="input(this.name)"/>
    <button name="answer(no)" text="Cancel" action="input(this.name)"/>
</panel>
```

Wh-questions differ from the other question types in that they are non-interactive widgets. Like the other non-interactive move representations, they simply consist of a natural language representation of the move on a label. So the move "?*x*.action(*x*)", corresponding to the English utterance "What do you want to do?" is represented as:

```
<panel columns="1">
    <label text="What do you want to do?"/>
</panel>
```

Linearization types for Thinlet output

We use both natural language phrases and GoDiS semantics when producing the Thinlet widgets, which is reflected in the linearization types. To the original linearization records in the speech and semantics grammars, we add a record row for XML output:

```
lincat Move, ... = SMove ** SSem ** XML;
```

The XML linearization type is defined in the XML resource module and consists of a string under the label `xml`:

```
oper XML       : Type = {xml : Str};
     xmlConcat : XML -> XML -> XML = \x1,x2 ->
                 {xml = x1.xml ++ x2.xml};
     xmlBegin  : Str -> Str -> XML = \elem,attrs ->
                 "<" ++ elem ++ attrs ++ ">";
     xmlEnd    : ...;
     xmlEmpty  : ...;
```

The XML resource module also defines operations for creating XML elements and attributes, which are then used by the Thinlet resource module in defining operations for creating widgets of different kinds:

```
oper panel  : XML -> XML = \xmlContents ->
              xmlConcat (xmlBegin "panel" (xmlAttr "columns" "1" ...))
             (xmlConcat xmlContents (xmlEnd "panel"));
     label  : Str -> XML = \text ->
              xmlEmpty "label" (xmlAttr "text" text ...);
     button : Str -> Str -> XML = \name,text ->
              xmlEmpty "button" (xmlAttr "name" name
                                 (xmlAttr "text" text ...));
```

Reuse of existing grammars

Note that we need both an English and a Swedish version of the Thinlet grammar, since the GUI consists of text as well as graphical objects. We do this in the same way as before by writing an incomplete concrete grammar in which the XML output is known, but the specific language is unknown. We reuse the concrete syntaxes from the spoken language interface, and from the semantics, for creating the Thinlet widgets:

```
incomplete concrete Godis_Thinlet_Incomplete of Godis_Abstract =
    open (Text = Godis_Speech_Incomplete),
         (Sem = Godis_Semantics),
         Resource_Thinlet in ...
```

We do not have any participant in the dialogue moves, since this is a system output-only grammar. The non-interactive moves are quite straightforward, such as the “quit” move:

```
lin quitMove = Text.quitMove ** Sem.quitMove **
    panel (label (Text.quitMove.s));
```

Questions are slightly more complicated. Yes/no-questions consist of a label and two buttons:

```
lin ynqQuestion ynq = Text.ynqQuestion ynq ** Sem.ynqQuestion ynq **
    panel (xmlConcat (label (ynq.sQue))
          (xmlConcat (button "answer(yes)" "Yes")
                     (button "answer(no)"  "No")));
```

Alternative questions consist of a number of buttons, plus an extra “Cancel” button:

```
lin altQ qs = Text.altQ qs ** Sem.altQ qs **
    panel (xmlConcat qs (button "answer(no)" "Cancel"));
```

Each alternative yes/no-question corresponds to a button:

```
lin ConsYNQ q qs = Text.ConsYNQ q qs ** Sem.ConsYNQ q qs **
    xmlConcat (button ((pp1 "answer" q).sem) (q.sInd)) qs;
```

Abstracting the GUI description language

In the grammar library we have chosen Thinlet as the GUI description language of our choice. But if we also want other similar description languages, we can rename and abstractify the resource module `Resource_Thinlet` into the *interface* module `Resource_GUI_Interface`. Then `Resource_GUI_Thinlet` would be an *instance* of the interface, and the GF/GoDiS concrete grammar module `Godis_GUI_Incomplete` would depend on two incomplete modules.

No changes necessary to the domain grammars

Note that everything that is needed for Thinlet grammars is found in the central GF/GoDiS grammar. This means that exactly the same domain-specific grammar can be used for GUI output or spoken output.

3.4.4 Integrated multimodality – utterances with click modality

We do not have to change the abstract syntax since no new categories or functions are defined. Definitions of the specific multimodal demonstrative expressions are left to the dialogue domain, since different domains and different sorts might have different associated utterances. E.g. to specify a place of departure one might have to say "from here" together with clicking on a map, but to specify a song to play, the associated utterance might be "this song".
The method for adding integrated multimodality in section 2.1.1 can be used on an *incomplete* concrete module as well as ordinary concrete modules. We use this fact to define a new incomplete module for the addition of a click modality to the concrete grammar.

```
incomplete concrete Godis_Click_Incomplete of Godis_Abstract =
    open (S = Godis_Speech_Incomplete),
         Resource_MultimodalPoint in ...
```

Note that we open the unimodal GoDiS grammar `Godis_Speech_Incomplete` (as the shortcut `S`), since the method tells us to preserve all unimodal information. In general, the definition of the function f looks like:

`lin` $f\ x_1 \ldots x_n$ `= S.`$f\ x_1 \ldots x_n$ `**` *(point/click information)*`;`

That is, we keep all information from the unimodal linearization, but add the click modality. This is only done for the *demonstrative* categories, and the categories which depend on demonstratives.

Demonstrative categories

The category which we use as demonstratives is the category `Ind` of individuals:

```
lincat Ind = Dem SS;
```

The categories which depend on indivuduals, i.e. having a function taking a demonstrative as argument, are also demonstratives:

```
lincat Proposition, Answer, Ask, ICM, Move, [Move],
       Utterance, SystemUtterance, UserUtterance = Dem SMove;
       YNQ, [YNQ], AltQ, Question                = Dem SQuestion;
```

Non-demonstrative categories

The rest of the categories have the same linearization types as in the unimodal grammar. There linearizations are simply reflections of the unimodal linearizations:

```
lin f x1...xn = S.f x1...xn;
    predWhQ = S.predWhQ;
```

Adding point/click information to demonstrative functions

A function taking only one demonstrative as argument, just propagates the click information.

```
lin pred1prop sort pred ind = S.pred1prop sort pred ind ** ind;
```

Note that due to record subtyping we can write `ind` instead of `mkPoint (ind.point)`. A demonstrative function taking no demonstrative arguments can be written like this:

```
lin requestMove m = S.requestMove m ** noPoint;
```

Finally, a function taking several demonstrative arguments has to concat the clicks of the arguments:

```
lin ConsMove m ms = S.ConsMove m ms ** concatPoint m ms;
```

Completing the click module

Finally we can complete the Click module with a specific target language by instantiating the phrases:

```
concrete Godis_Click_English of Godis_Abstract = Godis_Click_Incomplete
    with (Godis_Phrases_Interface = Godis_Phrases_English);
```

3.4.5 Strategies for improving speech recognition

With the given definitions, any dialogue move can follow or precede any other move. Such a liberal grammar is not good for the speech recognizer. What we need is a way of restricting the language model. Now, in the dialogue system we are focussing on, the following regular expression over dialogue move types captures the different possibilities of user utterances:

```
SingleMove |
Negative-ICM |
(YNAnswer | Positive-ICM)? (Request | Ask)? Answer*
```

By `Negative-ICM` we mean all ICM moves with negative action level, and by `Positive-ICM` we mean all ICM moves with positive action level. This regular expression can be directly encoded into the GF grammar with just a few minor changes:

- The category ICM has to be divided into two categories. Alternatively, ICM has to depend on Polarity:

  ```
  cat ICM Polarity;
  cat Polarity;
  fun Positive, Negative : Polarity;
  ```

- The function userUtterance has to be removed, and a number of other UserUtterance functions reflecting the regular expression have to be defined, among them something like the following:

  ```
  fun userRequest : MaybeYesnoICM -> Request -> [Answer];
  ```

This will restrict the user grammar, which in turn restricts the corpus that is generated in TALK deliverable D1.3 [Weilhammer et al., 2006].

Restricting the possible answers

The utterances can be restricted even further, by observing that for a given dialogue system, each action and question has a fixed number of follow-up questions – e.g. the action to play some music has two follow-up questions: which artist and which song. This means that only an artist answer and a song answer (in any order) should be allowed after a request to play some music. Also, even if the user does not request an action or asks a question, but only gives some answers, not all answers should be allowed. In some applications it might feel strange to give the same kind of answer several times in a row, regardless of which question the user is answering.

Our problem is how to encode such restrictions in the grammar. We give two possible solutions and hint at a third. But first we decide that the kind of answer is deduced from the sort – which is already done in the abstract syntax in section 3.3.2.

Solution 1: Encoding the restrictions in dependent types

We can change the types of questions (i.e. 1-place predicates) and actions to depend on a list of sorts:

```
cat Question [Sort]; Pred1 [Sort]; Action [Sort];
```

The restriction that a list of answer sorts can only be taken from a given list, can be encoded as Horn clauses:

$$\begin{array}{rl} & \mathit{AnswerList}([]) \\ \forall x,xs & \mathit{AnswerList}(xs) \rightarrow \mathit{AnswerList}(x:xs) \\ \forall xs,ys & \mathit{Select}(xs,ys) \wedge \mathit{AnswerList}(ys) \rightarrow \mathit{AnswerList}(xs) \end{array}$$

The second clause handles the case when a sort is not answered by the user. The last clause is when the user answers one of the possible sorts, which is then selected from the list of restrictions:

$$\begin{array}{rl} \forall x,xs & \mathit{Answer}(x) \rightarrow \mathit{Select}(x:xs,xs) \\ \forall x,xs,ys & \mathit{Select}(xs,ys) \rightarrow \mathit{Select}(x:xs,x:ys) \end{array}$$

These Horn clauses are straightforwardly translated into GF abstract syntax – the only things one have to remember are that the Horn predicates Answers and Select are GF categories, that variables are typed, that lists of sorts are constructed by `BaseSort` and `ConsSort`, and that each Horn clause gives rise to a unique GF function:

```
cat AnswerList [Sort]; Select [Sort] [Sort];

fun baseAnswers : AnswerList BaseSort;
    skipAnswers : (x:Sort) -> (xs:[Sort]) ->
                  AnswerList xs -> AnswerList (ConsSort x xs)
    consAnswers : (xs,ys:[Sort]) ->
                  Select xs ys -> AnswerList ys -> AnswerList xs
```

```
fun answerSelect  : (x:Sort) -> (xs:[Sort]) ->
                    Answer x -> Select (ConsSort x xs) xs
    recurseSelect : (x:Sort) -> (xs,ys:[Sort]) ->
                    Select xs ys -> Select (ConsSort x xs) (ConsSort x ys)
```

Now the function `userRequest`, defined above, can be redefined to include the sorts of the answers:

```
fun userRequest : (xs:[Sort]) ->
                  MaybeYesnoICM -> Request xs -> AnswerList xs;
```

This means that to each action and question we have to associate a list of sorts corresponding to the feasible follow-up answers:

```
fun playMusic : Action (ConsSort SongSort (ConsSort ArtistSort BaseSort));
```

How can this dependently typed grammar help speech recognition? There are two ways – either we can create a context-free speech recognition grammar obeying the restrictions, or we can use the grammar to generate a corpus from which a statistical language model can be trained.

It is possible to transform away the type dependencies since there is only a finite number of sort lists in the grammar – one for each action and predicate/wh-question. The simplest way is to instantiate the sorts and sort lists and incorporate them with the categories `Answer`, `Select` and `AnswerList`, thus giving a large number of context-free categories; `{Answer x}`, `{Select xs ys}` and `{AnswerList xs}` for all possibilities of `x`, `xs` and `ys`.

After having transformed away the type dependencies, the abstract syntax is a context-free grammar which then can be used to create a context-free speech recognition grammar, or to generate a corpus of syntax trees.

Solution 2: Restricting the generation of syntax trees

The second solution can be considered more of a hack than the first one. But on the other hand it is much simpler, since dependent types are not involved.

The idea is based on the fact that the tree generation commands in GF (called `gt` and `gr`) can be restricted by an incomplete syntax tree, meaning that all generated trees will be on the given form. The original domain grammar need not be altered, so the action `playMusic` for playing some music will just be an `Action`, without any type dependencies. However, we know that each user utterance for playing music will be on the form:

```
userRequest ? (requestAction playMusic) ?Answers
```

where the list of answers can be on one of the following forms:

```
BaseAnswer
(ConsAnswer SongSort BaseAnswer)
(ConsAnswer ArtistSort BaseAnswer)
(ConsAnswer SongSort (ConsAnswer ArtistSort BaseAnswer))
(ConsAnswer ArtistSort (ConsAnswer SongSort BaseAnswer))
```

This means that a corpus of all possible utterances for playing music can be generated by five GF commands for generating trees. This can of course be repeated for all actions and predicates/wh-questions in the domain. For a normal-sized domain there will be a huge number of GF tree generation commands that have to be invoked, but these can be generated automatically from the information saying which answer sorts each action and predicate/wh-question can take.

The solution will thus consist of some scripts that automatically generate a corpus of underspecified trees, which then can be fed into GF to generate an instantiated corpus. To each dialogue domain we only have to specify the kind of answers that are allowed to follow each predicate (in a wh-question) and action (in a request).

A drawback of this solution is that it can only be used when generating a corpus, but since TALK deliverable D1.3 [Weilhammer et al., 2006] shows that training an SLM gives better results than creating a speech recognition grammar, it is still a feasible solution.

Future solution 3: Transfer modules

The most general solution would probably be to use *transfer modules* to transform a non-restricted grammar into a restricted grammar, and put all restrictions on the actions and predicates/wh-questions in the transfer rules. However, this solution is still only theoretical since transfer modules are very experimental in GF and still under development.

Not implemented in the grammar library

None of the restrictions suggested in this section are implemented in the current version of the Multimodal Grammar Library, since there are different possible alternatives and we do not want to decide on one single solution.

3.5 Domain dependent grammars

In this section we describe the specifics of a dialogue domain. First we explain what is needed to create a new grammar for a given dialogue domain. Then we describe the specific details of our two example dialogue systems – the MP3 player and the calendar application.

The grammar library is designed to make it as easy as possible to create new dialogue grammars, which we hope is shown by our examples. Note that both examples are existing dialogue systems for which we have created new dialogue grammars, which can be seen as a kind of “stress test” for the core dialogue grammar described previously.

3.5.1 What is needed to describe a new domain?

Suppose we want to write a grammar for the new domain *Dom*. Then the following is a recipe of what kinds of things that have to be added to the core GoDiS grammar. We assume that the dialogue system is already specified, i.e. that the possible dialogue moves, sorts, individuals, etc. are known.

Ontologies

We have to decide which ontology grammars we will make use of in the domain. The domain grammar is extended by each of these grammars. Suppose that we are going to use the ontologies $O_1, \ldots, O_n$, then the abstract and concrete grammars start like follows, where $Lng \in \{\texttt{English}, \texttt{Swedish}\}$:

```
abstract Domain_Dom_Abstract =
         Godis_Abstract,
         Ontology_O1_Abstract,
         ...
         Ontology_On_Abstract ** ...

concrete Domain_Dom_Speech_Lng =
         Godis_Speech_Lng,
         Ontology_O1_Lng,
         ...
         Ontology_On_Lng ** ...
```

Sorts and individuals

The sorts and individuals are preferrably derived from the ontologies – i.e. for each main category *Cat* in an ontology *O*, we declare that *Cat*`Sort` is a sort:

```
fun CatSort : Sort;
```

We also have to give a coercion function from the elements of category *Cat* into the individuals of the sort *Cat*`Sort`:

```
fun indCat : Cat -> Ind CatSort;
```

In some cases we want to add elements to an ontology, or even define a new sort which does not depend on an ontology. In this case we simple enumerate the individuals:

```
fun i1, ..., im : Ind CatSort;
```

The sort *Cat*`Sort` only has to be given a linearization in the GoDiS semantics module, and this will be the GoDiS predicate that corresponds to the sort:

```
lin CatSort = "the_category_as_recognized_by_godis";
```

In all languages, the individuals are linearized as in the ontology:

```
lin indCat x = x;
```

Predicates

For each 0-place predicate *p* in the domain, associated with the sort *s*, define the GF constant *p*:

```
fun p : Pred0 s;
```

The linearization for the GoDiS semantics is straightforward:[11]

```
lin p = pp0 "p";
```

When linearizing to a language, we must decide which dialogue participant is allowed to state that *p* is true. The linearization should be as a proposition, e.g.:

```
lin p = sUser (variants{"it is true that p"; "p is true"});
```

A 1-place predicate *p* is also associated with a sort *s*, and both its abstract definition and the semantics are similar:

```
fun p : Pred1 s;
lin p = pp0 "p";
```

However, the linearization type for spoken utterances consists of three constituents as explained in section 3.3.2: *p* can be applied to an individual to form a proposition, *p* can be used as a wh-question, which in turn can be direct or indirect. This is done by giving three phrases, of which the propositional phrase consists of a "hole" where the individual will be put:

```
lin artists_songP = {who = System} **
                    sHole ["it is"] ["who has made the song"] **
                    sQue  ("who" ++ hasmadeV ++ optStr songN)
                          ["who made the song"];
```

Note that the record `system` says that the system can give answers ("it is ABBA who has made the song"), and the opposite participant (i.e. the user) can ask questions ("who has made the song"). The indirect question occurs when the system talks about the given question ("let's return to the question about who made the song").

Questions, answers and reports

In some cases there might be questions which have no corresponding answer, or answers without corresponding questions. We might prefer to add these as direct questions or answers:

```
fun listenToQ : Question;
lin listenToQ = pWhQ "listenTo";
```

[11] The operations `pp0`, `pp1`, and `pp2` are defined in the Prolog resource module and create Prolog terms taking zero, one or two arguments.

In the natural language grammars we have to give both the direct and the indirect question, together with the preferred participant:

```
lin listenToQ = {who = System} **
                sQue ["do you want to listen to radio or music"]
                     ["whether you want to listen to radio or music"];
```

Some kind of answers can be described syntactically as system reports, e.g. when the system fails to find an answer to a question it can be described as a report of a failure instead of an answer. This is left as a choice to the grammar writer.

Actions

Each action *a* in the dialogue system is added as a GF function, with the trivial linearization in the semantics:

```
fun a : Action;
lin a = pp0 "a";
```

There are two ways of uttering an action – either as a request from the user, or as a declarative when the system reports that the action is done:

```
lin a = sAct ["do a"] ["I have done a"];
```

Phrases

To simplify things, common phrases can be put in resource modules, which might be called `Domain_`*Dom*`_Phrases_`*Lng*, as described previously.

Adding multimodality

Parallel GUI multimodality is handled automatically by the core GoDiS grammar, but for the integrated "click" multimodality, some things have to be added. To the abstract syntax we have to add functions from points to individuals, for each of the multimodal "here-with-a-click" expressions. I.e. for each clickable *Cat* in an ontology, we add the function `this`*Cat*:

```
abstract Domain_Dom_Click_Abstract = Domain_Dom_Abstract ** {
  fun thisCat : Point -> Ind CatSort;
  ...
```

The concrete grammar module reuses the unimodal speech module:

```
concrete Domain_Dom_Click_Lng of Domain_Dom_Click_Abstract =
         Godis_Click_Lng,
         ..., Ontology_Oi_Lng, ... **
         open (S = Domain_Dom_Speech_Lng) in {
```

Each multimodal individual this*Cat* is defined as a demonstrative:

```
lin this Cat p = ss ["this cat"] ** mkPoint p;
```

The other individuals are not demonstratives, i.e. they have no point associated with them:

```
lin ind Cat x = S.ind Cat x ** noPoint;
    ...
    c_i = S.c_i ** noPoint;
```

Finally, predicates and actions are not changed at all:

```
lin act_i = S.act_i;
    pred_j = S.pred_j;
```

3.5.2 DJ GoDIS

Ontologies

The ontologies used in this domain are the following:

- Music – Artists, Albums and Songs
- Radio stations
- Numbers

Sorts and individuals

The Sorts reflects the ontologies: ArtistSort, AlbumSort, SongSort and StationSort (for radio stations). The PlaylistSort is currently so small and domain specific that it is not described in any ontology. The ontology of numbers is used to define IndexSort, the sort of playlist indices.

The Individuals are just the elements of the respective ontologies. A playlist index can be any natural number, or the special indices “next” and “previous”.

Predicates

The predicates that the user can ask for and the system find answers to are *path* (to find the search path to a song), *artists_song* (which artist has made a given song), *artists_album* (which artist has made a given album), *albums_by_artist* (which albums has a given artist made), *songs_by_artist* (which songs has a given artist made), and *current_song* (which is the song currently playing).

The predicates that the system asks for are *album*, *station*, *artist*, *song*, *playlist*, *itemAdd* (which item should be added to the playlist), *itemRem* (which item should be removed from the playlist), *groupToAdd* (which group should be added to the playlist), *song_artist* (which artist has made the song), and *what_to_play* (which song in the playlist should be played).

Questions and answers

There is one general question: “do you want to listen to music or the radio”, which is encoded as the GoDiS predicate *listenTo*.
There are three special answers, which are used when it is impossible to find a suitable answer to a question – *path_nomatch* (there is no matching search path), *artists_album_bestof* (the album is a compilation of many artists), and *albums_by_artist_nomatch* (there are no albums by the artist in the database).

Actions

There are in total 23 possible actions that can be requested, which range from playing, stopping and pausing, via fast forward and backward, and handling the volume control, to adding and removing items from the playlist. There are also top-level actions for restarting the dialogue, and selecting the player, the playlist or the radio.

Adding integrated multimodality

All sorts except numbers are clickable, meaning that we add the following functions:

```
fun thisArtist  : Point -> Ind ArtistSort;
    thisAlbum   : Point -> Ind AlbumSort;
    thisSong    : Point -> Ind SongSort;
    thisStation : Point -> Ind StationSort;
```

The different sorts can be referred to by different phrases, e.g. an artist can be referred to as “them”, “her”, “him”, etc., while an album or a song are better referred to as “this” or “that”:

```
lin thisArtist p = ss (variants{"them";"her";"him";["this artist"]}) **
                   mkPoint p;
    thisAlbum  p = ss (variants{"this";"that"} ++ optStr "album") **
                   mkPoint p;
```

3.5.3 Agenda-Talk

Ontologies

The ontologies used in this domain are the following:

- Time expressions
- Date expressions
- Event types
- Locations

Sorts and individuals

The Sorts reflects the ontologies: TimeSort, DateSort, EventSort and LocationSort. Furthermore, AMPM-Sort is defined together with the time expressions. The Individuals are just the elements of the respective ontologies.

There is also a meta-level Sort called InfoSort which is used when we want to specify another sort (date, time or location):

```
fun InfoSort : Sort;
    dateInfo, timeInfo, locationInfo : Ind InfoSort;
lin dateInfo     = ss ["date"];
    timeInfo     = ss ["time"];
    locationInfo = ss (variants{["place"];["location"]});
```

Finally, there is a BookingSort, consisting of lists of events and times:

```
fun BookingSort   : Sort;
    emptyBookings : Ind BookingSort;
    bookings      : [EventTime] -> Ind BookingSort;
lin emptyBookings  = ss ["no bookings"];
    bookings books = ss (["the following bookings :"] ++ books.s);
```

A booked event can have a specified time, or be without time information:

```
fun eventTime   : Event -> Time -> EventTime;
    eventNotime : Event -> EventTime;
lin eventTime e t = ss (e.s ++ t.s);
    eventNotime e = ss (e.s ++ ["unspecified time"]);
```

Predicates

The predicates that both dialogue participants can ask for are *store_start_time*, *store_duration_time*, *store_location* and *store_event*. When the user asks these questions, the idea is to find the starting time, duration time, location or event for an existing booking. The system however, can also ask questions about when, how long, where and what for new bookings.

There are two predicates that only the user can ask for, *todays_date* (what is today's date) and *bookings* (what are the scheduled events a given date).

The predicates that the system asks for are *newtime* (what time should the event be moved to), *olddate* (what date is the event currently), *newdate* (what date should the event be moved to) and *which_info* (which information is wrong).

There are also three system-only 0-place predicates: *usage* (giving information about the application), *take_down_event* (saying that a new event will be recorded), and *empty_event* (saying that there is nothing booked on the specified time).

Questions and answers

There is one general question: "do you want to add any more information", which is encoded as the GoDiS predicate *add_more_info*.

Actions

The following are the possible actions:

- adding and removing bookings – *add_event*, *delete_event*, *delete_current_event*;
- adding more information to a booking – *more_info*;
- changing information, date or time of a booking – *change_info*, *change_date*, *change_time*;
- checking existing bookings – *get_info*.

There is also a top-level action for restarting the dialogue.

Adding integrated multimodality

The sorts that are clickable are dates, times and booked events. For each of these sorts we add a demonstrative function `thisDate`, `thisTime` and `thisEvent`.
The different sorts can be referrred to by different phrases, e.g. a date can be referred to as "then", "this date" or "that date", while an event is referred to as "this", "that" or "this/that event".

3.6 The Edinburgh Town Info grammar

The *Town Info* GF grammar has been developed at UEDIN to cover the in-car information-seeking domain of the SACTI data collections and the baseline system with reinforcement learning, see Lemon et al. [2006] and TALK deliverable D4.2 [Lemon et al., 2005]. It has also been used to test the method presented in 2.1.1 for adding multimodality to an existing grammar.
There is currently not a GoDiS application for the *Town Info* grammar, since it is used with the DIPPER dialogue manager. Therefore it is not part of the GF/GoDiS grammar library, but the structure is very similar to the previous examples *DJ GoDiS* and *AgendaTalk* in section 3.5.

3.6.1 The unimodal grammar

Ontologies

The ontologies used in this domain are:

- Restaurants: cuisine type, price range, address, name
- Bars: bar type, price range, address, name
- Hotels: class, room type, address, name - Numbers

Sorts and Individuals

The sorts reflect the ontologies: RestaurantCuisineType, RestaurantPriceRange, RestaurantAddress, RestaurantName, BarType, BarPriceRange, BarAddress, BarName, HotelClass, HotelRoomType, HotelAddress, HotelName, and ChoiceNumber.

The individuals are elements of these respective ontologies.

Predicates

The predicates that the user can ask for are *choice_number* (to choose a particular entity and get its description), and the different task types (*town_info_hotels*, *town_info_bars*, *town_info_restaurants*).

The predicates the system asks for are the *cuisine_type*, *restaurant_price_range*, *bar_type* (e.g. a jazz bar or wine bar), *bar_price_range*, *hotel_class*, *hotel_room_type*, and *option_number* (which of the presented options should be described fully).

Actions

The user can request to change to a different task, ask the system to restart, repeat, or select a particular option.

3.6.2 Adding integrated multimodality

A proof-of-concept multimodal version of this grammar was obtained by following the steps presented in section 2.1.1.

A map location ("here") is now clickable, allowing the user to choose an option on the map by saying say "here [click]" etc.

The abstract grammar consists of the following multimodal rules:

```
fun p1      : Point;
    LocHere : Point -> Phrs;
    UttRule : Phrs -> Sentence;
    MMinput : Sentence -> MMsentence;
```

Using the multimodal conversion combinators, the linearization types become the following:

```
lincat Point      = Point;
       Phrs       = Dem SS;
       Sentence   = Dem SS;
       MMsentence = SS;
```

The linearizations also use the multimodal combinators:

```
lin p1        = mkPoint "p1";
    LocHere p = ss ["here"] ** mkPoint p;
    UttRule u = u;
    MMinput i = ss (i.s ++ ";" ++ i.point);
```

This enables parsing of simple multimodal expressions as follows:

```
> p -cat=MMsentence "here ; p1"
MMinput (UttRule (LocHere p1))
```

3.6.3 Coverage

Here are some examples showing the coverage of the current grammar (the "Bpm" prefixes are present because the base grammar is in fact compiled from a Business Process Model describing the domain):

```
> p "i need some indian food"
UttRule (UttRulePhrs23 I_need_some (Bpm_generalTypeRule_13
(Bpm_town_info_restaurants_cuisine_indian_Type_Rule
Bpm_town_info_restaurants_cuisine_indian))
(Bpm_spotting_restaurants_1 spot_restaurants_food))

> p "i um want a chinese meal and uhhh a cheap hotel please in the town center"
UttRule (ConsPhrs (UttRulePhrs7 Want A (restaurants_ask_food_type_1
(Bpm_restaurants_cuisine_chinese_Type_Rule Bpm_chinese))) (ConsPhrs (UttRulePhrs2
(Bpm_spotting_restaurants_1 spot_restaurants_meal)) (ConsPhrs (UttRulePhrs4 A
(hotels_ask_price_1 (Bpm_cheap_Type_Rule Bpm_cheap))) (ConsPhrs (UttRulePhrs2
(Bpm_spotting_hotels_1 spot_hotels_hotel)) (ConsPhrs (UttRulePhrs2
(hotels_ask_location_2 (Bpm_central_Type_Rule Bpm_town))) (ConsPhrs (UttRulePhrs2
(hotels_ask_location_2 (Bpm_central_Type_Rule Bpm_center))) BasePhrs))))))
```

There is a development set of 207 user utterances which were collected for building the enhanced language models for the baseline dialogue system of TALK deliverable D4.2 [Lemon et al., 2005].

The current *Town Info* grammar performs quite well on this test set at 91% (189 of 207 test sentences). This could be improved with further development time.

3.7 Summary

In this chapter we have described in detail the contents of the multimodal and multilingual GF/GoDiS grammar library, written in Grammatical Framework. The grammars in the library have been made multimodal by using the method described in chapter 2 in this deliverable.

The grammar library connects user and system utterances specified in GF with a dialogue system using the GoDiS dialogue manager. The library is designed for making is easy to add new dialogue domains, source languages, and input and output modalities. Currently the library consists of two dialogue domains, each with two source languages and three different modalities. The two domains are the calendar application *AgendaTalk*, and the MP3 player *DJ GoDiS*.

An additional GF grammar has been described, which has been made multimodal using the given method. The Edinburgh *Town Info* grammar is not part of the GF/GoDiS grammar library, but can be seen as a proof-of-concept of the generality of the method. Another additional grammar, the *Tram Demo* grammar,

was used as a pedagogical example when introducing the method in chapter 2 and has therefore not been described in this chapter.

By using the diversity of the GF module system, such as resource modules, incomplete modules, interfaces and instances, we have maximized sharing of common information between languages, modalities, ontologies and domains. This is done to make adding a new language, modality, ontology or domain as simple as possible.

Chapter 4

Summary and Conclusions

The ISU approach uses abstract representations for dialogue states and update rules which allow the generic characterisation of flexible dialogue strategies. This enables the same code for dialogue management techniques to be used for different natural languages and for different domains.

In this deliverable, we have shown that by using an abstract representation for grammars, we can further enable rapid porting of dialogue systems between languages, domains and modalities. The main tool in defining such grammars is Grammatical Framework (GF), which is used in collaboration by UGOT, UEDIN and UCAM for making ISU-based dialogue systems.

We have described two approaches to adding multimodality to unimodal dialogue systems and grammars. The first approach is to implement multimodality at the grammar level. We have given a language- and domain-independent method for how to add multimodal information to a unimodal GF grammar, thus simplifying the transition from a speech-only dialogue system to a multimodal one. The second approach is to implement multimodality at the level of the dialogue manager, which has been tried out in the ISU-based dialogue system DelfosNCL, developed by USEV.

The main part of the deliverable has been a detailed description of the multimodal and multilingual GF/GoDiS grammar library, written in Grammatical Framework. The grammar library connects user and system utterances specified in GF with a dialogue system using the ISU-based GoDiS dialogue manager. The library has been designed for making it easy to add new dialogue domains, source languages, and input and output modalities. Currently the library consists of two dialogue domains, each with two source languages and three different modalities. The two domains are the calendar application *AgendaTalk*, and the MP3 player *DJ GoDiS*.

Furthermore, two additional multimodal GF grammars have been described, which are created using the method for adding multimodality. They are not part of the GF/GoDiS grammar library since they are not part of a GoDiS dialogue system, but can be seen as proofs-of-concept of the generality of the method. The *Tram Demo* grammar, used by the UGOT Tram Information System (GOTTIS), has been used as a pedagogical example when introducing the method. The UEDIN *Town Info* grammar, used with the ISU-based DIPPER dialogue manager, has been used to test the method.

By using the diversity of the GF module system, such as resource modules, incomplete modules, interfaces and instances, we have maximized sharing of common information between languages, modalities, ontologies and domains. This has been done to make adding a new language, modality, ontology or domain as simple as possible.

Conclusions

We have presented techniques for incorporating multimodality into dialogue systems with grammars. We have distinguished two kinds of multimodality: parallel and integrated multimodality (multimodal fusion) and explored techniques for incorporating this both into grammars directly and by treating in terms of the interaction between grammar and dialogue management. We have shown a general technique for creating multimodal grammars from unimodal grammars.

While the kind of multimodality we handle is limited (selection by clicking and/or talking) we are pointing to general techniques that will enable to the rapid development of new multimodal dialogue systems in a way that will be generalizable to other kinds of multimodality as well.

Bibliography

Richard A. Bolt. Put-that-there: Voice and gesture at the graphics interface. *ACM SIGGRAPH Computer Graphics*, 14(3):262–270, July 1980.

Björn Bringert, Robin Cooper, Peter Ljunglöf, and Aarne Ranta. Development of multimodal and multilingual grammars: viability and motivation. Deliverable D1.2a, TALK Project, 2005. URL `http://www.talk-project.org/`.

Håkan Burden and Peter Ljunglöf. Parsing linear context-free rewriting systems. In *IWPT'05, 9th International Workshop on Parsing Technologies*, Vancouver, Canada, October 2005.

Michael Johnston. Unification-based multimodal parsing. In *Coling-ACL*, pages 624–630, 1998.

Michael Johnston and Srinivas Bangalore. Finite state multimodal parsing and understanding. In *Proceedings of the 18th conference on Computational linguistics*, pages 369–375, 2000.

Michael Johnston, Philip R. Cohen, David McGee, Sharon L. Oviatt, James A. Pitman, and Ira A. Smith. Unification-based multimodal integration. In *ACL*, pages 281–288, 1997.

Staffan Larsson. *Issue-based Dialogue Management*. PhD thesis, Göteborg University, Göteborg, Sweden, 2002.

Oliver Lemon, Kallirroi Georgila, James Henderson, and Matthew Stuttle. An ISU dialogue system exhibiting reinforcement learning of dialogue policies: generic slot-filling in the TALK in-car system. In *Proceedings of EACL*, page to appear, 2006.

Oliver Lemon, Kallirroi Georgila, and Matthew Stuttle. Showcase exhibiting reinforcement learning for dialogue strategies in the in-car domain. Deliverable D4.2, TALK Project, 2005. URL `http://www.talk-project.org/`.

Peter Ljunglöf. *Expressivity and Complexity of the Grammatical Framework*. PhD thesis, Göteborg University and Chalmers University of Technology, Gothenburg, Sweden, November 2004.

Peter Ljunglöf, Björn Bringert, Robin Cooper, Ann-Charlotte Forslund, David Hjelm, Rebecca Jonsson, Staffan Larsson, and Aarne Ranta. The TALK grammar library: an integration of GF with TrindiKit. Deliverable D1.1, TALK Project, 2005. URL `http://www.talk-project.org/`.

S.L̃. Oviatt, A. DeAngeli, and K. Kuhn. Integration and synchronization of input modes during multimodal human-computer interaction. In *Proceedings of Conference on Human Factors in Computing Systems, CHI '97*, New York, 1997. ACM Press.

Marint Pickering and Simon Garrod. Toward a mechanistic psychology of dialogue. *Behavioral and Brain Sciences*, 27(2):169–226, 2004.

José F. Quesada, Doroteo Torre, and Gabriel Amores. Design of a natural command language dialogue system. Siridus Project Deliverable D3.2, 2000.

Aarne Ranta. Grammatical Framework, a type-theoretical grammar formalism. *Journal of Functional Programming*, 14(2):145–189, 2004.

David Traum, Johan Bos, Robin Cooper, Staffan Larsson, Ian Lewin, Colin Matheson, and Massimo Poesio. A model of dialogue moves and information state revision. Trindi Project Deliverable D2.1, 1999.

Karl Weilhammer, Rebecca Jonson, Aarne Ranta, and Steve Young. Generation of language models using GF. Deliverable D1.3, TALK Project, 2006. URL `http://www.talk-project.org/`.

Appendix A

The Multimodal Grammar Library

A.1 Downloading the grammar library

The TALK Multimodal Grammar Library can be downloaded from

`http://www.ling.gu.se/projekt/talk/software/`

The distribution consists of a collection of GF grammar modules, distributed in the following directories:

Godis Core grammars for GoDiS-based dialogue systems

Ontology Grammars describing general ontologies

Domain The application domains MP3 (for the *DJ GoDiS* application) and Agenda (for the *AgendaTalk* application

Resource General GF resource grammars

The directories and grammar modules are described in more detail in chapter 3.

A.2 Installation instructions

First download and install Grammatical Framework. Source code, binaries and installation instructions can be found on the GF homepage:

`http://www.cs.chalmers.se/~aarne/GF/`

Set the search path to the GF library, and start GF from inside the directory of the grammar files:

- In csh, tcsh:

```
> setenv GF_LIB_PATH (path-to-GF)/lib
> (path-to-GF)/bin/gf
Welcome to Grammatical Framework, Version 2.4
...
```

- In bash:

```
$ export GF_LIB_PATH=(path-to-GF)/lib
$ (path-to-GF)/bin/gf
Welcome to Grammatical Framework, Version 2.4
...
```

If GF will be used on a regular basis, the `gf` binary should be added to the global search path and the environment variable `GF_LIB_PATH` should be set globally.

A.3 Testing the grammars

The grammars can be tested separately by loading them into GF. The relevant concrete syntax modules are:

> `Domain_`*Dom*`_`*Src*`_`*Lng*`.gf`, where *Dom* $\in$ {`MP3`, `Agenda`}, *Src* $\in$ {`Speech`, `Thinlet`, `Click`}, and *Lng* $\in$ {`English`, `Swedish`}.

The following is an example of the capabilities of the GF program. For more information about how to use GF, see the GF documentation. This example assumes we are testing the *DJ GoDiS* spoken language grammar, which of course can be replaced by any of the other grammars in the library.

1. Start GF in the directory where the grammars are located:

```
$ cd (path-to-grammar-library)/Domain/MP3/
$ gf
```

2. Load the source module(s) into GF:

```
> i -conversion=finite Domain_MP3_Semantics.gf
> i -conversion=finite Speech/Domain_MP3_Speech_English.gf
> i -conversion=finite Speech/Domain_MP3_Speech_Swedish.gf
```

   The option `-conversion=finite` compiles away finite dependent types, which are used as described in section 3.3.2. Without this option the parser returns too many parse trees, which have to be filtered by the GF command `pt -transform=solve`.

3. Select the English concrete grammar, and the starting category:

```
> sf -lang=Domain_MP3_Speech_English
> sf -cat=UserUtterance
```

4. Parse an English utterance:

```
> p -cfg "play like a virgin by madonna"
UserUtterance (ConsMove ...)
```

   The option `-cfg` selects an improved context-free parsing algorithm. The default parsing algorithm is overgenerating on grammars with dependent types, such as this one, and the resulting parse trees have to be filtered by `pt -transform=solve`.

5. Translate (i.e. parsing followed by linearization) from English to Swedish:

```
> p -cfg "play like a virgin by madonna" | l -all -lang=Domain_MP3_Speech_Swedish
spela like a virgin med madonna / ...
```

   The option `-all` shows all possible variants of linearizing a syntax term.

6. Translate from English to GoDiS dialogue moves:

```
> p -cfg "play like a virgin by madonna" | l -lang=Domain_MP3_Semantics
[request(play),answer(song(like_a_virgin)),answer(artist(madonna))]
```

7. Generate 5 random Swedish utterances:

```
> gr -number=5 | l -lang=Domain_MP3_Speech_Swedish
in the city med eagle eye cherry
rant radio
va
jag vill ändra balansen mitten tack
jag vill spela nummer tre tack
```

8. Quit GF:

```
> q
```