Dialogue Systems as Proof Editors*

Aarne Ranta
Dept. of Computing Science
Chalmers & Gothenburg University
S-41296 Gothenburg
aarne@cs.chalmers.se

Robin Cooper
Dept. of Linguistics
Gothenburg University
S-40530 Gothenburg
cooper@ling.gu.se

Abstract. This paper shows how a dialogue system for information-seeking dialogues can be implemented in a type-theory-based syntax editor, originally developed for editing mathematical proofs. The implementation gives a simple logical metatheory to such dialogue systems and also suggests new functions to them, e.g. a local undo operation. The method developed provides a logically based declarative way of implementing simple dialogue systems that is easy to port to new domains.

1 Dialogue systems for information seeking dialogue

In [5, 16], a distinction is made between command language dialogues, information seeking dialogues and negotiative dialogues. An information seeking dialogue is one where one of the agents (typically the system in a human-computer dialogue) asks a series of questions to which it needs to know the answer in order to carry out some task. Such dialogues normally begin (after initial greeting) with an exchange between the user and the system which establishes what task is to be carried out (e.g. search for information concerning a flight) and followed by a series of exchanges in which the system asks for information from the user needed in order to carry out the task (e.g. where the user wants to fly to, where the user wants to fly from, what date and time the user wishes to travel etc.). We will call this the *information gathering phase*. The dialogue normally ends with the carrying out of the task (e.g. the system informs the user of the details of a flight matching the user's answers to the questions) and a sign-off exchange ("bye"). We will follow the model and terminology for such dialogues developed in the EC project TRINDI [11].

The standing example we will use in this paper is the information gathering phase of a travel booking dialogue, such as the following:

*Work on this paper was supported in part by the European SIRIDUS project, IST-1999-10516. The authors are grateful to Peter Ljunglöf and Bengt Nordström for comments and to Carlos Gonzalía for help with Spanish grammar.

1. Where are you travelling from?
— From Paris.

2. Where do you want to go?
— To Stockholm.

3. By what mode of transport?
— By train.

4. In which class?
— Second.

5. When?
— Tomorrow.

This rigid-order question-answer dialogue can be compared with **filling a form**:

```
From: ____________________________
To: ______________________________
Mode of transport:__  Class: _____
Date: ____________________________
```

Even if seemingly the most boring mode of communication, the form already has more flexibility than a rigid-order dialogue: the customer may, at least, fill the form in any desired order, and also change what she has previously filled.

A dialogue system is often driven by a finite-state automaton which asks the questions in a certain order and expects to get answers to exactly the questions that have been asked. It is, of course, possible to build in more flexibility in the ordering of the questions by adding arcs to the finite state machine. But it is not straightforward to build a general dialogue strategy module that can easily be ported to different domains in this way. At the other end of the flexibility scale, there are dialogue systems based on the AI notion of planning and very powerful reasoning facilities [4]. While having the potential of producing high-quality dialogues, such systems run the risk of being inefficient (because of the complexity of the reasoning task), as well as impredictable and therefore difficult to maintain. The GoDiS system developed in the TRINDI project [6] is between these extreme ends of the flexibility scale. In GoDiS, an attempt was made to exploit the use of information states representing an agent's view of the state of the dialogue in order to allow such flexibility in a principled way. GoDiS keeps track of questions that the system plans to ask and questions that are currently under discussion. It will accept answers to questions that the system has not yet asked but has on its dialogue plan. In this paper we will relate this to the process of refinement in a proof editor and thus present a more proof-theoretic view of information seeking dialogues than has hitherto been suggested.

2 Proof editors

A proof editor is an interactive computer system for constructing proofs and other mathematical objects. The basic ideas date back to the interactive program editors of 1970's [18], and were extended to cover mathematical proofs in the 1980's. Today's proof editors can be roughly divided into two classes. In systems like Isabelle [12] and Coq [1], interaction is conceived as giving hints to an automatic theorem prover, by choices of **tactics**. In systems like ALF [9], interaction is conceived as direct manipulation of a proof object by stepwise **refinements** of it. It is this latter mode of proof editing that will serve as our model of dialogue. The actual implementation is carried out in the system GF (Grammatical Framework) [15], which extends a proof editor of the ALF type into a grammar formalism. GF, like ALF, has its theoretical basis in the **type theory** of Martin-Löf [10].

In a proof editor, the user is constructing an **object** of a given **type**. The construction process consists of a sequence of **refinements**, each of which gives one more bit of information, and of **undo** steps, cancelling earlier-made refinements. For instance, booking a travel can be formalized as the construction of an object of type `Booking`, by using the function

```
book :
  City -> City -> (v:Vehicle) -> Class v -> Date -> Booking.
```

In the beginning of the editing session, the user has the **incomplete object**

```
book [City] [City′] [Vehicle] [Class] [Date] :  Booking.
```

where the elements in brackets are **metavariables**, also called **subgoals**. Each metavariable has a type, which determines what refinements are possible for it. The type of the function `book` determines that the metavariables of the object at hand are typed as follows:

```
[City]    : City,
[City']   : City,
[Vehicle] : Vehicle,
[Class]   : Class [Vehicle],
[Date]    : Date.
```

Notice, in particular, that the type `Class` is a **dependent type**: what classes there are depends on the vehicle. The following **typing judgements** define what vehicles and classes there are:

```
Flight : Vehicle ;
Train  : Vehicle ;
Bus    : Vehicle ;

Business : Class Flight ;
```

```
Economy  : Class Flight ;
Second   : Class Train ;
First    : Class Train ;
NoClass  : Class Bus ;
```

We'll take a closer look at the refinement process later; an example first refinement step in the initial situation is

```
refine [City'] with Stockholm
```

which results in a more complete object

```
book [City] Stockholm [Vehicle] [Class] [Date] : Booking.
```

The notion of **reasoning** used in the proof editor consists of type checking, evaluation of λ-terms, and some constraint solving. Efficient methods have been developed for doing this [3], so that large and complex proofs can be handled.

3 Desirable features of dialogue systems

The following desiderata are from the Trindi Tick List [11], used as evaluation criteria for dialogue systems. We will cite them all, each with an example and possibly some comments. We set the machine's utterances in italic and the user's in roman type.

Q1 The system is sensitive to context.

> *When do you want to leave?*
> Tomorrow.
> *When do you want to return?*
> Two days later.

Here we have context-sensitivity in many senses: First, the **slot** that is being filled determines that it is the departure that takes place *tomorrow*, not the return. Second, the **environment** in which the dialogue takes place determines what date *tomorrow* is. Third, the **history** of preceding moves determines that *two days later* is two days after tomorrow. From the logical point of view, these three notions of context are quite different, and increasingly difficult to formalize.

Q2 The system accepts more information than requested.

> *Where do you want to go?*
> To Stockholm, tomorrow.

Two questions are answered at the same time, and the *when* question will not be repeated.

Q3 The system accepts an answer to a different question.

> *Where do you want to go?*
> I want to leave tomorrow.

The *when* question will not be repeated, but the *where* question will.

Q4 The system accepts an answer with incomplete information.

> *Where do you want to go?*
> To South America.

Even though the precise destination is not determined, the answer rules out London, Paris, etc.

Q5 Ambiguity is accepted.

> London.
> *London U.K. or London Ontario?*

This is not always easy to tell apart from Q4.

Q6 Negatively specified information is handled properly.

> *When do you want to leave?*
> Not today anyway.

The original purpose of this tick list item was to verify that the answer is *not* interpreted as the same as plain *today*, as some simple **keyword spotting** dialogue systems do.

Q7 The system can live with no information at all.

> *Where do you want to go?*
> —

The phone service should e.g. hang up or repeat the question instead of waiting for ever.

Q8 The input can be noisy.

> *Where do you want to go?*
> `ZXXXXXXXZZZZZZZKkK` to Stockholm you `#?/%/##`

The system should be robust enough to extract appropriate information in a noisy environment even if what can be perceived does not make a coherent phrase. This is often done by keyword or key-phrase spotting.

Q9 The user can initiate help dialogues.

> *How do you want to travel?*
> What alternatives are there?
> *Flight, train and bus.*

Help dialogues concern what the system is capable of.

Q10 The user can initiate non-help dialogues.

> *How do you want to travel?*
> Is there a train?
> *Yes.*

Non-help dialogues have to do with the state of the database rather than the capabilities of the system. For example, the question of whether the system has information about trains would be considered a help question whereas whether there is a train to a particular destination could be a non-help question about the state of the database addressed to a system which in general does have train information.

Q11 The system only asks appropriate follow-up questions.

> *How do you want to travel?*
> By train.
> *Which class?*
> Second.
> *When?*

If train is chosen as the means of transport, the question arises which class, but, if bus is chosen, the question does not arise:

> *How do you want to travel?*
> By bus.
> *When?*

Q12 The system can deal with inconsistent information.

> I want to leave tomorrow, Friday.
> *Tomorrow is Thursday. Do you mean Thursday or Friday?*

The system recognizes inconsistencies and prompts the user to clarify.

4 Interaction in a proof editor

Just like a dialogue system as viewed on the information state approach advocated in TRINDI, a proof editor has an **(information) state**, telling what object has been constructed so far and what is still missing. The proof editor state can be represented as a quadruple

```
State = (Term, Type, Subgoals, Constraints)
```

where the `Subgoals` field is a list of metavariables paired with their types,

```
Subgoals = [(Meta,Type)]
```

and the `Constraints` field is a list of pairs of terms that are required to be equal,

```
Constraints = [(Term,Term)]
```

(We use Haskell [17], the implementation language of GF, as presentation metalanguage.) This definition of state is somewhat simplified: actual proof editors also have a `Context` of bound variables, both globally and for each metavariable. We do not need contexts since our application has no variable-binding operations; but the type-theoretical notion of context has proved useful in modelling intricate context-dependencies, such as anaphoric references in a discourse [14].

A set of **commands** can be defined as functions from `State` to `State`. The interpretation of course presupposes an underlying **theory**, a set of typing judgements and definitions; but since the theory is not affected by the commands, we omit it in the following type signatures:

```
refineOne     :: (Meta,Term)   -> State -> State
refineMany    :: [(Meta,Term)] -> State -> State
refineAct     :: Term          -> State -> State
activate      :: Meta          -> State -> State
addConstraint :: [(Term,Term)] -> State -> State
localUndo     :: SubTermId     -> State -> State
```

The **semantics** of a command is the very state-transforming function it is defined as. This is defined by means of **type checking**, whose details we will not present here (see [8, 3] for type checking algorithms for type theory with metavariables). Intuitively, what the `refine*` commands do is replace metavariables by terms, which may themselves contain new metavariables. This can be done one at a time (`refineOne`) or with a whole list (`refineMany`). A refinement term that contains new metavariables is the primary way in which the user can give partial information to the system.

In the `refineAct` command, the metavariable is not specified by the user: one of them (e.g. the first in the list) at a time can be considered as the **active** subgoal, and the refinement concerns that one. A separate `activate` command (e.g. a mouse click on a metavariable) can be used to change the active subgoal. Notice that the active metavariable in the proof editor plays the same role as the **current question** in a dialogue system.

The `addConstraint` command adds a constraint to the state. This constraint is an equation between two terms, which can contain metavariables. In some cases, a constraint can lead to the solution of a metavariable by ruling out all alternatives but one. Besides refinement with incomplete terms, adding constraints is a means of giving partial information. Unlike refinement, which is strictly positive (choose a path in the choice tree), constraints can also be negative (rule out some paths). They are, however, always **decidable**, in the sense that, for any instantiation of metavariables, the correctness of the resulting constraint can be mechanically checked.

The `localUndo` command uses a subterm identifier (= some way of identifying a subterm) to delete a part of the already constructed term, and to replace it by a metavariable. This corresponds to a situation where the customer has already said she will go to Paris and later changes her mind. It is one of the advantages of proof editors that this operation is definable: in some systems, where the current information state is a set of sentences rather than a structured term, it is hard to tell what effects a local undo should have. This is a well-known problem in the literature on **belief revision**.

The use of dependent types implies that local undo is not always possible: for instance, if you want to undo `Flight` without first undoing `Business`, `Flight` gets automatically restored as a derived refinement. There is a way to improve this rather rude behaviour without sacrificing type-correctness: when removing a subterm, also remove all those subterms that it would create as instances of unification [8]. This requires a more complex structure of state than the one we have been discussing.

In addition to state-changing commands, there are commands to display information, quit, and so on. Thus we can define

```
showActive    :: State -> Meta
showResult    :: State -> Term
displayMenu   :: Meta  -> [Term]
askIfPossible :: Meta  -> Term -> Bool
quit          :: IO ()
```

We will relate these commands to the Trindi tick list in a table below.

Observe, finally, that a state-transforming command may **fail**: for instance, a refinement may be inconsistent with the expected type of the subgoal, or there may be no metavariable there corresponding to the refinement. In these cases, the simplest way to define the command is that it does not alter the state. The system can still respond by an informative error message. In Haskell this is conveniently programmed by using a so-called error monad: the value type is not just `State`, but either a state or a message.

5 An example proof editor session

Let us follow through a proof editor session for constructing a booking. We use a simplified version of GF, in this first example without the natural language support. After each command, we display the new term, and the active metavariable; the active metavariable should be compared to the question that the system presents to the user. We start from the initial state, where nothing has been chosen.

```
book [City] [City'] [Vehicle] [Class] [Date] : Booking
? [City] : City

> refineAct Paris

book Paris [City'] [Vehicle] [Class] [Date] : Booking
? [City'] : City
```

The user's first move was a completely regular answer to the current question.

```
> refineOne ([Date],Today)

book Paris [City'] [Vehicle] [Class] Wed : Booking
? [City'] : City
```

Now the user answers a different question. Moreover, the answer is the **indexical** expression `Today`, a global constant that the system replaces by its definition `Wed` (type theory also has a form of judgement for such definitions).

```
> refineMany [([City],Here),([Vehicle],Train)]

book Paris Gothenburg Train [Class] Wed : Booking
? [Class] : Class Train
```

This was an example of a multiple refinement. It remains to give the class.

```
> refineAct Business

book Paris Gothenburg Train [Class] Wed : Booking
! Train <> Flight
? [Class] : Class Train
```

Since `Business` is a class for flights but not for trains, the system displays an **unsolvable constraint** and refuses to make the refinement. Instead, the previous question is repeated.

```
> displayMenu [Class]

Second First

> localUndo Vehicle

book Paris Gothenburg [Vehicle] [Class] Wed : Booking
? [Vehicle] : Vehicle
```

The user can now go on by first selecting `Flight` and then `Business`, thus completing the booking. She can also make use of the dependent type system: refining `[Class]` with `Business` will automatically replace `[Vehicle]` with `Flight` as a **derived refinement**:

```
> refineOne ([Class], Business)

book Paris Gothenburg Flight Business Wed : Booking
Booking complete.
```

Alternatively, refining `[Vehicle]` with `Bus` only leaves room for one possible value of `[Class]`, which is automatically filled in as a **unique refinement**:

```
> refineOne ([Vehicle], Bus)

book Paris Gothenburg Bus NoClass Wed : Booking
Booking complete.
```

6 Proof editors evaluated as dialogue systems

Let us now return to the Trindi Tick List and see what requirements the proof editor fulfils. Those marked with *soon* will be mended later in this paper.

Q	requirement	ALF/GF	how
1	context-sensitivity	yes	first meta; global constants
2	more information	yes	`refineMany`
3	different information	yes	`refineOne (Meta,t)`
4	less information	yes	refine with incomplete term
5	ambiguity	soon	parse input + display alternatives
6	negative information	soon	real parsing, `addConstraint`
7	no answer	no	set time limit?
8	noise	soon	parse with fault tolerance
9	help menu	yes	`displayMenu`
10	non-help menu	yes	`askIfPossible`
11	appropriate follow-up	yes	dependent types
12	inconsistency	yes	unsolvable constraints

The results compare fairly well with the evaluation of commercial dialogue systems in [11].

In addition to the TRINDI tick list, we have shown that at least two supplementary requirements can readily be fulfilled by the proof editor model of dialogue systems:

Q13 The system can infer answers to other questions.

> *How do you want to travel?*
> In business class.
> *OK, flight in business class.*

Q14 Earlier choices can be cancelled.

> Well, actually I don't want to go to Paris after all.
> *Where do you want to go then?*

7 Extending the proof editor: an experiment

Even if proof editors perform outstandingly as dialogue systems by the Trindi criteria, they mostly lack one fundamental feature: they use a special language for interaction, a written formal language. We will now show how to replace the formal language with natural language. We have also completed this move by using a speech synthesizer [2] to produce the system's reactions and echo the user's moves (so that the demo can be played with voice only). We have not yet investigated the use of speech recognition for user input.

The Grammatical Framework GF is an implementation of type theory, reminiscent of ALF, but extended by **concrete syntax** [15]. The place of an ALF theory is in GF taken by a **grammar**, which is a pair

```
Grammar = (Abstract, Concrete)
```

where the `Abstract` component is a theory in the sense of ALF. The `Concrete` component defines a mapping from type-theoretical objects into strings, which is also usable as a **parser** from strings to type theory. For instance, the object

```
book Paris Gothenburg Train Second Wed
```

is mapped to the string

```
from Paris to Gothenburg by train in the second class
on Wednesday
```

by an English concrete grammar. One and the same abstract syntax can be given alternative concrete syntaxes, so that the same object is mapped to the Spanish string

```
de Paris a Gotemburgo, tren en segunda clase el miercoles.
```

by a Spanish concrete grammar.

GF has a proof-editing command language similar to that of ALF. With the introduction of concrete syntax, GF extends the refine commands by **refinement by parsing**,

```
refineActParse :: String -> State -> [State].
```

The `String` input is translated into a `Term` by a parser, and there can be many results, among which the user may then choose: a simple model for this is that the value of the command is a list of (alternative) states rather than a single state. Of course, parsing in the ordinary linguistic sense is completed by type checking, so that the set of alternatives can be narrowed down by excluding states with unsolvable constraints.

GF also has a concept of **fault-tolerant parsing**, which makes it possible to parse input that is not in the grammar. Fault tolerance consists of two aspects:

> Morphological ill-formedness can be forgiven, so that e.g. *I wants to take an train* is recognized.
>
> Unknown words, i.e. words not occurring in the grammar, can be ignored by the parser. Thus *%$& to Stockholm ZZZZKkk* can be parsed as *to Stockholm*.

While the first of these aspects has been in GF from the beginning and mostly works in the expected way, the second aspect is experimental and not so useful as one might expect. It does make the parser robust in presence of meaningless noise, but it does not take away words that occur somewhere else in the grammar.

So it seems we now have solutions to the requirements Q5 (ambiguity) and Q8 (noise), at least beginnings of solutions. While noise tolerance could be improved by using other techniques that those proper to GF, dealing with ambiguities deserves a closer look. The problem is that, what GF already tolerates, is an ambiguous refinement *to a determined metavariable.* This is because a GF parser is a function

```
parseTerm :: String -> [Term].
```

What we have had to add is a parser

```
parseRefinement :: String -> [(Meta,Term)],
```

which also tells what question is being answered. More generally, the prototype needs a parser for potentially multiple refinements,

```
parseRefinements :: String -> [[(Meta,Term)]],
```

and, even more generally, for arbitrary potentially ambiguous commands,

```
parseCommand :: String -> [Command],
```

since we have to be able to distinguish between e.g. refinements and additions of constraints without explicit keywords, as well as to parse commands like `quit` and `displayMenu`.

The interaction in the propotype takes place entirely in natural language, which is seen on a terminal and can also be played through the speech synthesizer. An example dialogue is the following:

```
From where?
> from here of course
What destination?
> to Paris ; tomorrow
How?
> alternatives ?
bus, flight, train
How?
> in the first class
from Gothenburg to Paris train in the first class on Thursday
> good bye then
bye
```

The system uses type checking to infer what questions are being answered by each answer. Multiple refinements (in this prototype) must be separated by semicolons. When the object is complete, the system summarizes the situation.

As can be seen from the example, the system has not only to parse commands and update the state, but also to print (and utter) messages as a function of command and state:

```
message :: Command -> State -> String
```

For instance, the message corresponding to the `displayMenu` command is the list of possible refinements to the active metavariable. This list is printed through the concrete syntax of the language in which interaction is taking place. The response to the `quit` command is an equivalent of “goodbye” in the interaction language. The response to an impossible selection is a phrase equivalent to “sorry, the choice is impossible”. And, in any state where a question is active, the system utters that question.

The theoretical principle implicit in the type of the `message` function is that messages can be generated from the same information as is needed to change the state. Furthermore, all messages are specified in GF grammars, so that they can be easily modified and ported to new languages. This completes the claim that the definition of new dialogue systems is fully declarative.

The implementation of the "dialogue engine" as an additional module of the existing GF implementation (60 modules, 10K lines of code) required about 150 lines of Haskell code. In addition, there is GF code for basic grammars, about 20 lines each, defining the messages generated for each command. The English and Spanish travel agency grammars are about 40 lines of code each.

8 Open problems and future directions

An important issue is **scalability**: how would the proof editor work in a real-world travel agency, with thousands of travel destinations and connections? All these objects should certainly be loaded dynamically from a database, rather than encoded statically in a GF grammar. One way of doing this is to use the interface between GF terms and XML objects [7]. Other database formats could be usable, as well, but would require more thought, since XML has the advantage of having the notion of DTD (Document Type Declaration), which imposes a type system on data objects analogous to the inductive types of type theory. Since all interaction is driven by types, it is important that all data objects are correctly typed. What more is required of the database is some linguistic information, such as names of cities in the languages in which the dialogue system is expected to function.

Using the dialogue system as an alternative interface to any given GF grammar, e.g. for constructing proofs, is an appealing idea, but the results so far obtained are very modest: one must first solve the general problem of expressing mathematical objects in speech without access to a visual medium (cf. [13]).

We have studied a way of relating proof editing techniques to straightforward information seeking dialogues. In work subsequent to TRINDI more complex dialogue plan languages have been developed to deal with menu structures rather than simple form filling. It seems likely that the use of dependent types to make questions depend on other questions could be exploited in this connection but we have not investigated this.

In the SIRIDUS project we are not investigating notions of **negotiative dialogue**. In negotiative dialogues more than one answer to a question can be entertained simultaneously as alternatives and the agents need to collaborate on making a choice. For example, one might want to keep open two options as to where you fly from, e.g. Malmö or Copenhagen, or there may be several different departure times the user wishes to consider making a final choice depending on the price, time of arrival and so on. In the proof editor context this would involve assigning alternative values to metavariables and keeping track of the consequences of the different choices. While in general proof theoretic techniques should be able to assist with putting such negotiations on a firm theoretical foundation, we have not yet investigated how this should be done and it may involve extensions of the proof editors we have been considering.

9 Conclusion

Going through an existing evaluation scheme (the Trindi Tick List) and building a prototype upon the proof editor GF confirms that proof editors provide a possible theoretical model of dialogue systems, as well as an implementation platform for them.

The arguments in favour of this treatment of dialogue systems are both theoretical and practical:

> Proof editors based on type theory have a well-understood theory of interaction.
>
> The dialogue system framework can be extended by new functionalities such as local undo and inferred answers.
>
> Specific dialogue systems can be defined in a concise and theoretically well-founded way that exploits formal inference.

The proof-editing model gives support to information-seeking dialogues. We have not yet investigated its applicability to other forms of dialogue, such as negotiative.

References

[1] Coq Homepage. `http://pauillac.inria.fr/coq/`, 1999.

[2] A. W. Black. The Festival Speech Synthesis System. `http://www.cstr.ed.ac.uk/projects/festival/`, 1999.

[3] C. Coquand and T. Coquand. Structured type theory. In *Workshop on Logical Frameworkds and Meta-languages*, Paris, France, 1999.

[4] R. W. Smith and D. R. Hipp. *Spoken Natural Language Dialogue Systems.* Oxford University Press, New York and Oxford, 1994.

[5] J. Gabriel Amores and J. F. Quesada. Dialogue moves in natural command languages. Siridus deliverable D1.1, SRI International, Cambridge., 2000.

[6] S. Larsson and D. Traum. Information state and dialogue management in the trindi dialogue move engine toolkit. *Natural Language Engineering. Special Issue on Best Practice in Spoken Language Dialogue Systems Engineering*, To appear.

[7] V. Lux, M. Dymetman and A. Ranta. XML and multilingual document authoring: Convergent trends. In *Proc. Computational Linguistics COLING, Saarbrücken, Germany*, pages 243–249. International Committee on Computational Linguistics, 2000.

[8] L. Magnusson. An algorithm for checking incomplete proof objects in type theory with localization and unification. In S. Berardi and M. Coppo, editors, *Types For Proofs and Programs*, number 1158 in Lecture Notes in Computer Science, pages 183–200. Springer-Verlag, 1996.

[9] L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, number 806 in LNCS, pages 213–237, Nijmegen, 1994. Springer-Verlag.

[10] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.

[11] S. Larsson, I. Lewin, C. Matheson, P. Bohlin, J. Bos and D. Milward. Survey of existing interactive systems. Trindi deliverable D1.3, Gothenburg University, Gothenburg, 1999.

[12] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[13] T. V. Raman. *Audio System for Technical Readings.* PhD thesis, Cornell University, 1994.

[14] A. Ranta. *Type Theoretical Grammar.* Oxford University Press, 1994.

[15] A. Ranta. Grammatical Framework Homepage. `http://www.cs.chalmers.se/~aarne/GF/`, 2000.

[16] I. Lewin, S. Ericsson and C. J. Rupp. Dialogue moves in negotiative dialogues. Siridus deliverable D1.2, SRI International, Cambridge, 2000.

[17] S. Thompson. *Haskell the Craft of FUnctional Programming.* Addison-Wesley, Harlow etc., 2nd edition, 1999.

[18] G. Kahn, B. Lang, V. Donzeau-Gouge, G. Huet and J. J. Levy. A structure-oriented program editor: a first step towards computer assisted programming. In *International Computing Symposium (ICS'75)*, 1975.