Formalizing the Dialogue Move Engine

Peter Ljunglöf

Dept. of Computer Science
Chalmers University of Technology
412 96 Göteborg, Sweden
peb@cs.chalmers.se

Abstract

In this paper we present a calculus for reasoning mathematically about rule-based dialogue systems – so called *dialogue move engines* developed in the TRINDI project. The calculus is similar to term rewriting systems and dynamic logic. It is defined using monads, which are used for describing programming languages, and in functional programming to capture computations with side-effects.

1. Introduction

In this paper we present a calculus for reasoning mathematically about rule-based dialogue systems – so called *dialogue move engines* developed in the TRINDI, SDS and INDI projects[1] (Bohlin et al, 1999; Traum et al, 1999). The calculus is similar to term rewriting systems (Visser and Benaissa, 1998) and dynamic logic (Harel, 1984). It is defined using monads, which are used for describing programming languages, and in functional programming to capture computations with side-effects (Moggi, 1991; Wadler, 1995). In the end we show how the calculus can be used to prove properties of a dialogue system.

1.1. Preliminaries

Since we are only interested in the dialogue manager part of a dialogue system, we assume that there exist good translations between utterances and dialogue moves. Without loss of generality we can then assume that the dialogue participants communicate using dialogue moves.

As a simplification we assume that the dialogue is serial – that the participants make their utterances one after another and that they never interrupt each other. Another simplification is that each utterance can be translated into a time-ordered list of dialogue moves, thus forgetting about overlapping sub-utterances and so on.

In the discussion at the end we will try to argue that these simplifications do not induce severe limitations on the strength of the framework.

1.2. Notational conventions

In this paper we use a lot of terminology taken from programming languages and type theory. For those not familiar with our way of writing things, here are some informal explanations.

We write $a \in A$ to say that the object a is of the type A. The basic type constructors we are going to use are $\times$, $\rightarrow$ and $[]$. Given two types A and B, $A \times B$ is the type of pairs of A and B, $A \rightarrow B$ is the type of functions from A to B, and $[A]$ is the type of lists of type A.

There are some standard operations and predicates on lists which we will use – the *delete* and *add* operations deletes and adds elements to a list, the *append* operation concatenates two lists, and the *member* predicate is a sequential member checking predicate, binding the second argument to each element of the list. We will also use the standard way of using lists to represent backtracking – computations that can fail or return several results – with the empty list representing failure (Wadler, 1985).

2. Defining the dialogue move engine

A dialogue move engine (DME for short) consists of a description of *i*) what the information state (infostate for short) looks like, *ii*) what kinds of dialogue moves there are and *iii*) how they are applied to the infostate, *iv*) a collection of update rules on the infostate, and *v*) an update algorithm which defines how the rules are used to update the infostate.

Parallel with the formalization, we introduce an example DME to illustrate the principles. This is a small subset of the information-seeking DME used in the GoDiS system (Traum et al, 1999), but it is general enough for the purposes of this paper.

2.1. The information state

The *information state* is seen as a representation of an agent's current knowledge, especially the part that change during the dialogue. In this formalism the infostate is a type *IS*. For the example DME we will use a record with the fields shown in table 1 below, where *plan* is a list of things to do in the future, *bel* is a list of beliefs, *qud* is a list of questions currently under discussion, and *lm* is a list of the dialogue moves that the other participant just uttered. We do not further define propositions and questions as the formalization is independent of which notion of proposition or

$$\begin{aligned} \mathit{plan} &\in [\mathit{Move}] \\ \mathit{bel} &\in [\mathit{Proposition}] \\ \mathit{qud} &\in [\mathit{Question}] \\ \mathit{lm} &\in [\mathit{Move}] \end{aligned}$$

Table 1: The information state used in the example

[1]TRINDI (Task Oriented Instructional Dialogue), EC Project LE4-8314. SDS (Swedish Dialogue Systems), NUTEK/HSFR Language Technology Project F1472/1997. INDI (Information Exchange in Dialogue), Riksbankens Jubileumsfond 1997-0134.

r_1 : *integrate_question*	r_2 : *integrate_answer*	r_3 : *answer_question*	s : *select_move*
conditions	**conditions**	**conditions**	**conditions**
member(lm,Q)	*member(lm,A)*	*member(qud,Q)*	*member(plan,M)*
is_question(Q)	*is_proposition(A)*	*member(bel,A)*	**effects**
effects	*member(qud,Q)*	*is_answer_to(Q,A)*	*delete(plan,M)*
delete(lm,Q)	*is_answer_to(Q,A)*	**effects**	***select****(M)*
add(qud,Q)	**effects**	*delete(qud,Q)*	
	delete(lm,A)	*add(plan,inform(A))*	
	delete(qud,Q)		
	add(bel,A)		

Table 2: The update rules used in the example

question is chosen. In this simple example we have just two kinds of *dialogue moves* – to *ask* a question and to *inform* of a fact, represented as a proposition.

We view a dialogue move as a basic type, so we need to know how to incorporate an utterance, represented as a sequence of moves, into the infostate. This is done with the function *apply* $\in$ *IS*$\times$[*Move*] $\rightarrow$ *IS* which updates the infostate with a list of moves. In our example the *apply* function simply adds the moves to the end of the *lm* list.

2.2. Update rules

An *update rule* specifies an update on the infostate (called the effect), which is guarded by a condition – if the condition holds, the effect can be applied. The effect may also have a side effect: it can select one or several dialogue moves to be performed. The conditions and effects are composed by combining basic operations on the elements of the infostate. The update rules of our example are listed in table 2 above. The first two rules interprets the user's last move – if it was a question, it will be added as a question under discussion, and if it was an answer to a question currently under discussion it will be added as a belief. The third rule answers a question under discussion, if the system knows the answer, and the fourth rule selects the first move on the plan to be uttered to the user. Since this last rule selects a dialogue move to be performed, we call it a *selection rule*.

More formally we can say that an update rule is a function that given an infostate, returns either a failure if the condition fails, or the different results of the effect applied to the infostate. This gives us *rule* $\in$ *Rule* where *Rule* $=$ *IS* $\rightarrow$ [*IS*$\times$[*Move*]].

2.3. The update algorithm

The *update algorithm* defines how these rules should be applied to an infostate; that is, given an infostate, the update algorithm updates the infostate and selects a list of moves to perform. This suggests that the update algorithm is a function *update* $\in$ *IS* $\rightarrow$ *IS*$\times$[*Move*].

The naive algorithm is to check the rules in order, and as soon as a rule applies update the infostate accordingly and then repeat the algorithm until there are no rules that apply.

But if we use this naive algorithm on our example rules, all the moves that are in the *plan* will be selected at once – this is maybe not immediate from the definitions, but can be proved using the formalism we will introduce later. Since we want it to just say one thing at each turn, we have to change the algorithm to first apply the rules $r_1 \ldots r_3$ until this can no longer be done, and then apply the rule s once, selecting only one move.

With these definitions we can define the *dialogue move engine* to be a function that, given a list of dialogue moves uttered by the user, applies them to the infostate, and then updates the infostate with the update algorithm, selecting new moves to perform during the updating. We now finally have a function *dme* $\in$ *IS*$\times$[*Move*] $\rightarrow$ *IS*$\times$[*Move*], with the very simple definition *dme* $=$ *update* $\circ$ *apply*.

3. A calculus of update rules

In this section we introduce a calculus for the update algorithm and show that this can be used to define the update rules themselves. The calculus is similar to term rewriting systems (Visser and Benaissa, 1998) and dynamic logic (Harel, 1984), with the main exceptions being that the rules also has the ability to communicate to the outer world by selecting dialogue moves to perform, and all the operators are deterministic.

We have two trivial rules and three basic operators that make new rules out of old ones:

- The *identity* rule 1 always succeeds without affecting the infostate and without selecting any moves.
- The *failure* rule 0 always fails.
- The *sequential composition* $r\,;r'$ of two rules first applies r, and if that succeeds, applies r' to the result of r The composition selects all the moves selected by either r or r'. It fails if either r or r' fails. Composition has 1 as an identity and 0 as a zero, which gives the laws $1\,;r = r\,;1 = r$ and $0\,;r = r\,;0 = 0$.
- The *deterministic choice* $r+r'$ first applies r, and only if that fails it applies r'. Choice has 0 as an identity and 1 as a left zero, giving the laws $0+r = r+0 = r$ and $1+r = 1$ (but not necessarily equal to $r+1$).
- The *repetition* r^* applies r and if that succeeds it executes r^* on the result (concatenating the selected moves). If r fails, it succeeds leaving the infostate unchanged and selecting nothing. The repetition can be unfolded using the other operators: $r^* = (r\,;r^*)+1$.

$$
\begin{aligned}
r_1 &= \exists q \in \mathit{lm}.\, \mathit{is\_question}(q)\,;\, \mathit{delete}(\mathit{lm}, q)\,;\, \mathit{add}(\mathit{qud}, q) \\
r_2 &= \exists a \in \mathit{lm}.\, \exists q \in \mathit{qud}.\, \mathit{is\_proposition}(a)\,;\, \mathit{is\_answer\_to}(q, a)\,;\, \mathit{delete}(\mathit{lm}, a)\,;\, \mathit{delete}(\mathit{qud}, q)\,;\, \mathit{add}(\mathit{bel}, a) \\
r_3 &= \exists q \in \mathit{qud}.\, \exists a \in \mathit{bel}.\, \mathit{is\_answer\_to}(q, a)\,;\, \mathit{delete}(\mathit{qud}, q)\,;\, \mathit{add}(\mathit{plan}, \mathit{inform}(a)) \\
s &= \exists m \in \mathit{plan}.\, \mathit{delete}(\mathit{plan}, m)\,;\, \mathbf{\mathit{select}}(m)
\end{aligned}
$$

Table 3: Formal definitions of the update rules of the example

With these definitions the update algorithm of our example can be defined as $(r_1 + r_2 + r_3)^*\,;s$. This suggests that the update algorithm is just a very complicated update rule. But this definition of the update algorithm is not completely correct; the type of the *update* function does not correspond to the type of the update rules. The main difference is that the update rules can fail, which the update algorithm is not allowed to. But a correctly defined update algorithm will never fail, which means that the list of results it returns will be non-empty. Then we can use the standard list function $\mathit{head} \in [A] \to A$, which gives the first item in a list, to extract the result we want. This gives for our example the resulting function $\mathit{update} = \mathit{head}((r_1 + r_2 + r_3)^*\,;s)$.

3.1. Defining the update rules

Now it turns out that we can use the calculus to define the update rules themselves. To apply an update rule we first check the conditions, and if they hold we can apply the effects. Both the conditions and the effects are ordered – we apply them in the order they are written. This means that an update rule is just a sequential composition of more basic rules, the individual conditions and effects. There is just one thing that needs to be taken care of – the special *member* predicate which introduces some kind of choice depending on the elements of the first argument. For that purpose we introduce the operator $\exists x \in A.\, r(x)$, where A is a list and $r(x)$ is a rule whenever x is an element in A. The idea is that if $A = [a_1, a_2, \ldots, a_n]$ when the rule is invoked, then $\exists x \in A.\, r(x) = r(a_1) + r(a_2) + \cdots + r(a_n)$.

Another addition is to add the special selection rule $\mathbf{\mathit{select}}(m)$, which leaves the infostate unchanged and selects the single move m. With these additions to our calculus, we can define the update rules of our example as in table 3 above. (We still have to give definitions of the basic rules of course).

4. Interpreting the calculus

Monads are standard tools in functional programming for capturing computations with side-effects, and they are also used in denotational semantics for defining programming languages (Wadler, 1995; Moggi, 1991). Here we are going to use them to give a precise definition of our calculus.

4.1. Introducing monads

The standard example of a monad is the type constructor [] which takes any type A and gives back $[A]$, the type of lists of objects of type A. A monad M is a type constructor with two operations: $\mathit{return} \in A \to M(A)$ and $\mathit{bind} \in M(A) \times (A \to M(A)) \to M(B)$, which also satisfy three identity and associativity laws. Some monads also are equipped with a zero element $0 \in M(A)$, and a plus operation $(+) \in M(A) \times M(A) \to M(A)$, which in turn satisfy a couple of other laws.

An example of a monad is the state monad $\mathit{SM}(A) = \mathit{IS} \to \mathit{IS} \times A$, with the definitions $\mathit{bind}(f, k) = \lambda s.\, k(a, s')$ where $(s', a) = f(s)$, and $\mathit{return}(a, s) = (s, a)$. Another example is the monad of lists $[A]$ which is also a monad with zero and plus; where *return* returns a singleton list, 0 is the empty list, and $l + l'$ concatenates the lists l and l'. The bind operation sends each element of the first list to the second function, concatenating the results, and can be defined by cases as $\mathit{bind}([], k) = []$ and $\mathit{bind}(a{:}l, k) = \mathit{append}(k(a), \mathit{bind}(l, k))$.

The list monad is often used to represent backtracking, which we will also do here. We can also combine monads – e.g. combining the two monads above gives us the backtrackable state monad $\mathit{BSM}(A) = \mathit{IS} \to [\mathit{IS} \times A]$, which is a monad with zero and plus.

4.2. A dialogue monad

The backtrackable state monad *BSM* gives us a way to define the rules and operators of our calculus, since the type *Rule* of update rules is just an instance of $\mathit{BSM}([\mathit{Move}])$. The 0 rule and the $(+)$ operator are exactly the same as 0 and $(+)$ for the monad. The identity and selection rules can be defined as $1 = \mathit{return}([])$ and $\mathbf{\mathit{select}}(m) = \mathit{return}([m])$ respecively. Sequentiation simply becomes $r\,;r' = \mathit{append}^M(r, r')$, where append^M is concatenation of lists lifted to the *BSM* monad,[2] and repetition is defined by the unfolding equation $r^* = (r\,;r^*) + 1$. For the $(\exists)$ operator we have to use the fact that the *BSM* monad is a function that takes an infostate and gives a list as result: $\exists x \in f.\, r(x) = \lambda s.\, \mathit{bind}(f(s), r)$.[3]

Apart from giving us a precise definition of the calculus of update rules, it also gives us all the properties of monads, like the associativity and identity laws. These laws are free for us to use when we want to prove statements about, or to rewrite our dialogue move engine to a more efficient one.

[2] The precise definition of the lifting of a function f to a monad is $f^M(r, r') = \mathit{bind}(r, \lambda m.\, \mathit{bind}(r', \lambda m'.\, \mathit{return}(f(m, m'))))$.

[3] Observe that the *bind* used here is the one in the list monad, not in the *BSM* monad. The field label f is seen as the function on the infostate that gives the current value of the field f.

5. Proving properties of the DME

If we have defined a collection of update rules together with an update algorithm, we may want to show that some desirable properties hold for this dialogue system. Most important are to show that the system terminates, always succeeds and always produces some moves to utter to the user, but there are also other interesting properties.

5.1. Termination

To prove that the update algorithm terminates for every given input, we have to show that all repetitions always terminate. For this we can use induction on some parts of the information state. In our example we can do induction on the total length of the *lm* and *qud* lists (to be more precise, $n = 2|lm| + |qud|$), and notice that when any of the rules $r_1 \ldots r_3$ is applied, the total length decreases (actually, it's the number n that decreases). This means that $(r_1 + r_2 + r_3)^*$ cannot continue forever, since the *lm* and *qud* lists will finally be empty.

5.2. Non-failure

Since the repetition r^* always succeeds if it terminates, the only thing we have to prove for our example is that the selection rule s always succeeds. In our example case there is a possibility that the *plan* at some point gets empty, so we have to change the update rules in some way – e.g. by replacing s with $s + \textbf{\textit{select}}(m)$ in the algorithm, where m is some default move.

5.3. Productivity

To show that the system is productive we have to show that, whenever it terminates and succeeds, it executes a ***select*** rule. In our example it is easily shown just by looking at the s rule.

We may also want to show that the system does not select too many dialogue moves at the same time (thus giving the user the opportunity to interrupt with e.g. clarifying questions). In our first naive definition of the update algorithm, the system selected all the moves that was on the plan, but in the second algorithm the system selects only one move at the time.

5.4. Other properties

Another interesting property is that the system is efficient. There could be some of the basic conditions or effects that take time to execute (e.g. the *is_answer_to* predicate which probably has to call a theorem prover). We might want to show that the system never calls such a predicate more than, say 5 times. In our example system, the *is_answer_to* condition is called a number of times which in the worst case can be in the order of $|qud| \cdot |bel|$, which in turn means that the system should probably have to be optimized in some way.

A final example of a property of a well-designed dialogue move engine is that two rules are commutative with respect to the choice operator, i.e. $r + r' = r' + r$. The reason why this is a good property is that if a dialogue system is on the form $(r_1 + r_2 + \cdots + r_n)^*$, where each pair of rules r_i, r_j commute, then that system can be implemented asynchronously, executing each rule r_i in parallel.

6. Discussion and future work

In this paper we have introduced a calculus for building and reasoning about dialogue move engines. With the use of a simple example we have defined the basic constructors of the calculus.

In the beginning we introduced some simplifications on the dialogue system, and we will now try to argue that they can be accounted for. The limitations were that the dialogue is serial, and that each utterance can be translated into an ordered list of dialogue moves. But that the participants talk at the same time or interrupt each other can be coded using special arguments to the dialogue moves, as can the overlapping of moves, so these are not real limitations.

If one wants to work with something other than *lists* of moves, e.g. sets or partially ordered collections, one can redefine the $(\,;\,)$ and *return* operations in an appropriate manner. (Which means that one uses another monad in place of the list monad). In the same way one can use another definition of the $(+)$ operation, as long as it still obeys the monadic laws – e.g. one may want the choice to look at the current infostate before it decides which of the rules to try.

This is very much work in progress. It remains to show that the framework can be used in real-world problems, where the infostate is much more complicated and there are more than four update rules. One possible research issue would be to see if the framework can be used to model the dialogue behaviour of a system. Possibly the calculus can be used to prove desired properties of the system as a whole – e.g. that it in the end always gives a relevant answer to a question, or that it fulfills given orders.

7. References

P. Bohlin, R. Cooper, E. Engdahl and S. Larsson. 1999. Information states and dialogue move engines. Gothenburg Papers in Computational Linguistics GPCL 99-1. URL *http://www.ling.gu.se/publications/*

D. Harel. 1984. Dynamic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, vol. II*. Reidel.

E. Moggi. 1991. Notions of computation and monads. *Information and Computation*, 93(1).

D. Traum, J. Bos, R. Cooper, S. Larsson, I. Lewin, C. Matheson and M. Poesio. 1999. A model of dialogue moves and information state revision. Trindi Deliverable D2.1. URL *http://www.ling.gu.se/research/projects/trindi/*

P. Wadler. 1985. How to replace failure by a list of successes. In *2nd Symposium on Functional Languages and Computer Architecure*. Lecture Notes in Computer Science LNCS 273. Springer Verlag.

P. Wadler. 1995. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*. Lecture Notes in Computer Science LNCS 925. Springer Verlag.

E. Visser and Z. Benaissa. 1998. A core language for rewriting. In C. Kirchner and H. Kirchner, editors, *2nd International Workshop on Rewriting Logic and its Applications*. Electronic Notes in Theoretical Computer Science 15. Elsevier.