Towards a basis for human-computer dialogue

Bengt Nordström
Chalmers, Göteborg

AIM/DTP Shonan Village, Sept 2011

Background

There is a notion of correctness in human-computer (and many human-human) dialogues which can be captured by dependent types.

Example of a human-human dialogue:

A: What do you want to do?

eat sleep work swim sail ...

B: I want to eat

A: Do you want to eat at home?

restaurant home your place ...

B: No, in a restaurant.

A: Fine, I would like to eat in a japanese restaurant

japanese swedish french korean ...

B: What about okonomiyaki?

A: OK

The result of the dialogue is the construction of a mathematical object

eat restaurant (japanese, okonomiyaki)

An essential part of the dialogue is the possibility of changing the object being constructed:

A: I want to eat Swedish food

B: OK, let's have surströmming at home

We have now changed the object:

eat restaurant (japanese, okonomiyaki)

➡

eat home surströmming

Example of a human-computer dialogue

A travel agency:

plane | rental car | hotel | boat

When you click on one of these you get some other alternatives.

plane

from:	to:	nr of persons:	leaving date	arrival date

hotel

where:	nr of persons:	leaving date	arrival date

In both these cases the answer to one question decides the shape of the rest of the dialogue:

- If you know that you are talking about ordering a hotel, then it doesn't make sense asking about the departure city.
- If you are filling in a date, the year decides how many days there are in February and different months has different number of days.

So, the shape (type) of something depends on the value of something else.

Example of a datatype for a dialogue:

```
Place = data
      home restaurant forest : Place
Food : Place -> Set
Food home = SwedishFood
Food restaurant = (x : Nationality, NationalFood x)
NationalFood : Country -> Set
NationalFood japan = JapaneseFood
NationalFood sweden = SwedishFood
NationalFood korea = KoreanFood
Food forest = Forestfood
Action =
  data   eat : (where : Place) (food: Food(where)) -> Action
       sleep: Place -> Action
       swim: From -> To -> Manner -> Action
       work: Place -> What -> Action
       sail : From -> To -> Passengers -> Action
```

Some words to be explained:

- Syntactical correctness of a dialogue
- Top-down vs. bottom-up dialogue
- User driven vs. system-driven dialogue
- sequential vs. random access

Goal

We want to model dialogue systems in which a human interacts with a computer to build an object.

From Wikipedia:
A **dialogue system** is a computer system intended to converse with a human, with a coherent structure. Dialogue systems have employed text, speech, graphics, haptics, gestures and other modes for communication on both the input and output channel.

Here I will ignore the problems of speech recognition and generation and abstract over the mode of interaction.

A dialogue is seen as the building of an object.

The human fills in information which the computer needs. Typical examples are:

- Reservation system for trains, concerts, calendars, rooms, medical doctors, ...
- Navigation systems
- Control center for TV, radio, mp3-players, ...
- Editors for controlled languages like
  - restaurant reviews
  - recipies
  - description of pharmaceuticals
  - description of items for sale
- structure oriented editors for programs and formal proofs

Typical structure of a dialogue system

speech input

| speech recognition

textual input

| parsing

syntax tree

| semantic interpretation

semantic representation

| dialogue management

system response content

| generation

system response utterance

| speech synthesis

system output

Checking correctness

In all these systems there is a notion of syntactic correctness, which should be checked immediately after each input of the user.

We will use a dependent type system to express the syntactical correctness.

Some choices:

- **System driven dialogue**: The system decides the order to fill in the details.
- **User driven dialogue**: The user decides.
  - **sequential access**: up, down, left, right...
  - **random access**: any part of the object can be changed.

Objects are treated as directed graphs

In principle, we have a graph like:

$$q_1 = c_1 \; p_1 \ldots \; p_n$$

$$\vdots$$

$$q_n = c_n \; r_1 \ldots \; r_m$$

where p, q and r are placeholders and c are functional constants.

System driven dialogue

The commands to the system is a list of (functional) constants:

c1, …, cn

each command replaces the leftmost placeholder.
Here, each constant has a fixed arity

User driven dialogue

The commands to the system is a list of commands of the form:

$$q = c\ q_1 \ldots q_n$$

The command replaces the placeholder q.

The commands to the system is a list of commands of the form:

$$q = c\ q_1 \ldots q_n$$

Bottom-up:
The placeholder q is new.

Top-down:
The placeholder q is already in use.

Sharing:
Some of the placeholders q_i are in use.

A simple example:

We want to define the constant 2 : Nat and we assume that we have the constant 0 : Nat and s : Nat -> Nat:

```
two :: Nat            -- declare the type of two
two := s q1           -- the system can deduce that q1 : Nat
q1  := s q2           -- the system can deduce that q2 : Nat
q2  := 0
```

How to introduce Nat, s and 0?

```
Nat :: Set
z    :: Nat
s    :: q1 -> q2     -- the system deduces that q1, q2 are types
q1  := Nat
q2  := Nat
```

The set of expressions are:

Thick expressions		**Thin expressions**
e,t :: = t -> t'	function types	e,t :: = q -> q'
| (t, t’)	cartesian product	| (q, q’)
| e, e’	pair	| q, q’
| (c e)	application	| (c q)
| q	place holder	| q
| c	constant	| c

Each thick expression can be represented as a list of thin expressions

Overview

Starting from a state with expressions with place holders, like:

    c1 : t1; c1 = q1; ...; cn : tn

we want to use a series of commands to build up a new state

c1 : t1; c1 = e1; ...; cn : tn; c1 = en

It is not necessary that all constants have a definiens.

But in the end all holes must be filled in (place holders must be defined).

Commands

q := e	Refine the value of the place holder q, e is a thin expression.
c :: e	Introduce a new constant with its type.
q :: e	Introduce a new placeholder and its type. This is only used in bottom-up editing. The type e must not be a functional type

Type checking

To type check a command of the shape

q := e

we always look up the type of q. It is an invariant of the system that all place holders have an expected type.

Then we type check e:

Type checking an application

```
q  :  Q         in G
c  :  C         in G
C = A ->B   in G
B = Q         in G
-----------------------------------------------------------
G, q := c q'  |-  G ;  q':A;  q = c q'
```

Explanation:

In order to type check the command q := c q' in G we:

- look up the type of q in G, call it Q.
- look up the type of c in G, it has to have the shape A ->B
- check that B = Q in G

Then we can update the state G with

- q' : A   (this update must be consistent with G)
- q = c q'   (and make sure that no cycles are introduced)

Type checking a constant/placeholder

q : Q in G
c : C in G
Q = C in G

G, q := c |- G ; q = c

Explanation:
In order to type check the command q := c in G we:

- check that the types of q and c are equal in G

Then we can update the state G with

- q = c   (this update must be consistent with G)

Type checking a pair

q : Q   in G
Q = (R, S) in G

G, q := r,s |- G ; q = r,s; r : R; s : S

Explanation:
In order to type check the command q := r,s in G we:

- Lookup the type of q in G. It has to have the shape (R,S).

Then we can update the state G with

- q = r, s   (this update must be consistent with G)
- r : R   (and make sure that no cycles are introduced)
- s : S

Type checking a type

q : Type   in G

G, q := r -> s |- G ; q = r -> s; r : Type; s : Type

Explanation:
In order to type check the command q := r -> s in G we:

- check that the type of q is Type in G

Then we can update the state G with

- q = r -> s   (this update must be consistent with G)
- r : Type   (and make sure that no cycles are introduced)
- t : Type

Consistent updates

The updates must be consistent in the sense that:

- No cycles are introduced
- An update of the shape $G; q : A$ is only possible if q does not have a type in G or q already has the type A in G.

Summary:

The user issues commands of the form

- q := e   refine a place holder
- q :: e   introduce a place holder

The expressions e are thin expressions of the form

```
e,t  :: =  q -> q'
          | (q, q’)
          | q, q’
          | (c q)
          | q
          | c
```

Summary (cont'd):

Expressions are represented as directed graphs with placeholders as names for subexpressions.

This makes it possible to build objects

- top-down
- bottom-up
- a combination of these

This is absolutely essential when trying to mimic human dialogues.

Applications

A way to structure a dialogue system:

The generic editor (handling objects with dependent types) works as a framework for different dialogue systems.

Further work:

Extend the language of objects and types to:

- dependent types
- some notion of a decidable subset type
- strategies for under-specified information (the system cannot deduce what placeholder to fill in) (Peter Ljunglöf)
- specification of concrete syntax

Thanks to:

- Aarne Ranta, Computer Science, Chalmers
- Björn Bringert, Google, London
- Peter Ljunglöv, Computer Science, University of Gothenburg
- Robin Cooper, Linguistics, University of Gothenburg

Snail!

Little by little climb up -

Mt. Fuji

- Issa