Understanding Information Update in Questionnaires

Jan van Eijck, Tijs van der Storm
CWI, Amsterdam

Abstract

Questionnaires are an important medium for collecting information in diverse areas of society (scientific surveys, tax filing, auditing guidance etc.). We are interested in a domain-specific language (DSL) to automatically generate questionnaire software from declarative specifications. This note describes an important aspect of the semantics of such a DSL: what goes on when users fill out a form? The formalism is based on the epistemic notion of information update and has a wide range of applications. It provides a formal interpretation for query forms, and for the process of answering such forms. The attractiveness of the approach is in the fact that asking questions, providing partial answers to questions, and providing full answers to questions, are all modeled by the same mechanism of constraining a range of possibilities.

Keywords: Domain-specific languages, questionnaires, semantics, epistemic logic.
2000 MSC: 03B42, 03B70, 03C13.

Although Paul Klint was trained as a mathematician, he dislikes formalization for its own sake. Paul is attracted to the practical side of computer science and software engineering, and his impatience with theory is understandable. All too often, in his view, theorizing is used as an excuse for inaction. Still, sometimes, a modest amount of formalization leads to better implementations, and this is when theory starts to be interesting for Paul.

1. Introduction

Questionnaires play a crucial role in society. Examples of domains where questionnaires are used include: scientific surveys [1], tax forms [8], checklists [5], emergency report forms [14] and auditing guidance material [3]. While questionnaires are traditionally – and sometimes still – presented on paper, the use of software-based questionnaires is wide-spread. The user answers questions presented as interactive forms, which may conditionally unfold to show more questions.

We're interested in automating the construction of questionnaire software through the use of a domain-specific language (DSL) [13]. Based on high-level, declarative descriptions of a questionnaire, much of the graphical user-interface (GUI) and back-end data handling can be automatically generated. As a result, questionnaires are easier to understand, change and validate.

The idea of using a DSL for creating questionnaires has received some attention in software engineering research (e.g., [1]). Most prominently however, is the Blaise language [2] which was designed by statisticians in the Netherlands during the 80s. Blaise can be used to define screens with typed widgets, enablement conditions, screen transitions (possibily conditionally) and the data model the form corresponds to. This language continues to be in use to this day.

Although Blaise is widely used, we're aiming for a language that has wider applicability, outside the domain of statistical surveys. Moreover, many types of questionnaires (e.g., tax forms, check-lists, emergency registration forms etc.) are subject to strict rules and regulations. It is therefore important that we can

URL: `jve@cwi.nl` (Jan van Eijck), `storm@cwi.nl` (Tijs van der Storm)

reason about the questionnaires themselves and about the way they are filled out. In this paper we formalize one aspect of a possible language for questionnaires, namely a precise semantics for modeling the process of filling out a questionnaire. This semantics is based on information update as known in epistemic logic.

Out starting point is the following question: What is the simplest possible way to look at what goes on when people provide information in some constrained format? It turns out that both posing a question in a fixed format and answering such a question can be viewed as forms of information updating, in a general sense of constraining a function. We propose a formalism that, because of its generality, has a wide range of applications.

The basic intuition is that information growth corresponds to elimination of possibilities, as in epistemic logic [7, 4]. Instead of possible worlds, we will use a more abstract space P of possibilities.

2. The Lattice of Constraint Functions

Let A be some finite alphabet. The set of all names constructed from A is A^*. Let P be some finite set. Think of this as a space of possibilities that can be constrained if information about what is possible gets added. For instance the set of possibilities $P = \{1, 2, 3, \text{true}, \text{false}\}$ allows names in A^* to be bound to 1, 2, 3, true or false. A *constraint function* is a function in $A^* \to \mathcal{P}(P)$. A constraint function assigns to each name in A^* a subset of the space of possibilities. For instance, given the definition of P above, a constraint function could restrict the set of possibilities of name x to the set $\{1, 2, 3\}$.

Intuitively, there are two dimensions of information growth about constraint functions: 1) Domain extension: the space of name fields gets extended; 2) Value constraint: values get more constrained. If one assumes that all name fields initially have unconstrained values, one can get by with just value constraint.

Moreover, the class of constraint functions, for given A and P, can be seen to be a lattice, with top $\top$ and bottom $\bot$ given by:

- $\top := \lambda x \mapsto P$,
- $\bot := \lambda x \mapsto \emptyset$.

$\top$ is the function that assigns to any name the full space of possibilities, and $\bot$ is the (overconstrained) function that assigns to any name the impossible set $\emptyset$. The lattice operations are given by:

- $f \sqcup f' := \lambda x \mapsto f(x) \cup f'(x)$,
- $f \sqcap f' := \lambda x \mapsto f(x) \cap f'(x)$.

The lattice ordering $\sqsubseteq$ is given by: $f \sqsubseteq f'$ iff $\forall x \in A^* : f(x) \subseteq f'(x)$.

3. Constraint Syntax

Assume a total ordering $<$ on the space of possibilities P. Use $p \leq q$ for $p < q$ or $p = q$. Let minValue be the minimum of P and maxValue its maximum. Basic constraints and propositional logic of constraints (assume p, q range over P, x ranges over A^*) is defined in the following grammar:

$$\phi \quad ::= \quad p < q \mid p \leq q \mid p \in x \mid \neg\phi \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \mid (\phi \leftrightarrow \phi).$$

The expression $p \in x$ effectively means that x is currently in the set of possibilities of x. In the example above, we assumed x to be constrained to $\{1, 2, 3\}$, so we have $1 \in x$, but not true $\in x$. Together with the ordering and the boolean connectives this allows us to express conditional update actions.

The language of update actions is then defined as follows:

$$\begin{array}{rcl} \pi & ::= & \text{skip} \mid x := C \mid \text{ if } \phi \text{ then } \pi \text{ else } \pi \mid \pi; \pi \mid \text{ while } \phi \text{ do } \pi \\ C & ::= & \lambda p \mapsto \phi \end{array}$$

The *skip* action changes nothing. The assignment statement updates the space of possibilities with a constraint property C so that after the update, ϕ is true for all possibilities related to x.

4. Constraint Semantics

Every constraint is interpreted as a boolean on constraint functions, as follows.

$$
\begin{array}{rcl}
[\![p < q]\!]_f & \iff & p < q \\
[\![p \leq q]\!]_f & \iff & p \leq q \\
[\![p \in x]\!]_f & \iff & p \in f(x) \\
[\![\neg\phi]\!]_f & \iff & \text{not } [\![\phi]\!]_f \\
[\![\phi_1 \wedge \phi_2]\!]_f & \iff & [\![\phi_1]\!]_f \text{ and } [\![\phi_2]\!]_f \\
[\![\phi_1 \vee \phi_2]\!]_f & \iff & [\![\phi_1]\!]_f \text{ or } [\![\phi_2]\!]_f \\
[\![\phi_1 \rightarrow \phi_2]\!]_f & \iff & \text{either not } [\![\phi_1]\!]_f, \text{ or } [\![\phi_2]\!]_f \\
[\![\phi_1 \leftrightarrow \phi_2]\!]_f & \iff & [\![\phi_1]\!]_f \text{ iff } [\![\phi_2]\!]_f.
\end{array}
$$

The key definition is the third one, which states that p is a possible value of x under the constraint function f if f would map x to a set which includes p.

Constraint properties are interpreted as subsets of P, as follows (where ϕ^p_q is name substitution of p by q):

$$[\![\lambda p \mapsto \phi]\!]_f \quad := \quad \{q \in P \mid [\![\phi^p_q]\!]_f\}.$$

A constraint property returns the set of values that make the constraint ϕ true under a constraint function f.

Every update action is interpreted as a relation on constraint functions, as follows:

$$
\begin{array}{rcl}
{}_f[\![\text{ skip }]\!]_{f'} & \iff & f = f' \\
{}_f[\![x := C]\!]_{f'} & \iff & \forall y \in A^* : \text{ if } x = y \text{ then } f'(y) = f(y) \cap [\![C]\!]_f \\
 & & \quad \text{otherwise } f'(y) = f(y) \\
{}_f[\![\text{ if } \phi \text{ then } \pi_1 \text{ else } \pi_2]\!]_{f'} & \iff & \text{if } [\![\phi]\!]_f \text{ then } {}_f[\![\pi_1]\!]_{f'} \text{ else } {}_f[\![\pi_2]\!]_{f'} \\
{}_f[\![\pi_1; \pi_2]\!]_{f'} & \iff & \exists f'' : {}_f[\![\pi_1]\!]_{f''} \text{ and } {}_{f''}[\![\pi_2]\!]_{f'} \\
{}_f[\![\text{ while } \phi \text{ do } \pi]\!]_{f'} & \iff & \text{either } [\![\neg\phi]\!]_f \text{ and } f = f', \\
 & & \text{or } \exists f'' : {}_f[\![\pi]\!]_{f''} \text{ and } {}_{f''}[\![\text{ while } \phi \text{ do } \pi]\!]_{f'}.
\end{array}
$$

Note that assignment to x is defined using intersection on the possible values for x.

The information provided by an update action is defined as the constraint function that results if the action is performed on the state of no information:

$$\text{info}(\pi) = f \text{ where } {}_\top[\![\pi]\!]_f.$$

5. Examples

To do examples, we assume that there are canonical mappings from datatypes to the space P. E.g., Booleans are encoded as $E_B = \{p_{true}, p_{false}\} \subset P$, characters are encoded as $E_C = \{p_a, \ldots, p_z\} \subset P$, character strings (up to some fixed size) are directly encoded as members of a subset Q of P, and so on.

Thus, we can think of a question or query as an action on constraint functions. The query action does two things: 1) It fixes a name in A^* for the value of the relevant constraint; 2) it fixes an encoding for the possible answers, as a subset of P.

Simple examples can now be formulated as follows:

- "What is your name?" Fixes a name $n \in A^*$ and fixes the possible values of the answer as a subset *Names* of P. Corresponding update: $n :=$ *Names*.

```
type Name = String
type CF a = Name → [a]

fullRange ::  (Enum a, Bounded a) ⇒ [a]
fullRange = [minBound..maxBound]

top :: (Enum a, Bounded a) ⇒ CF a
top = λ _ → fullRange

bottom :: CF a
bottom = λ _ → []

join :: (Eq a, Enum a, Bounded a) ⇒ CF a → CF a → CF a
join f g x = union (f x) (g x)

meet :: (Eq a, Enum a, Bounded a) ⇒ CF a → CF a → CF a
meet f g x = intersect (f x) (g x)
```

Figure 1: Constraint functions

- "What is your age?" Fixes a name $a \in A^*$ and fixes the possible values of the answer as a subset $Age = \{p_0, \dots, p_{120}\}$ of P. Corresponding update: $a := \{p_0, \dots, p_{120}\}$.
- "Are you male or female?" Fixes a name $g \in A^*$ and fixes the possible values of the answer as a subset $Gender = \{p_m, p_f\}$ of P. Corresponding update: $g := \{p_m, p_f\}$.

Answering such questions are updates too:

- "My name is *Jan*", in a context where the name query is interpreted as $n := Names$, is interpreted as $n := \{p_{\text{jan}}\}$, with $p_{\text{jan}} \in Names$.
- "I am male", in a context where the gender query is interpreted as $g := Gender$, is interpreted as $g := \{p_m\}$.

All of these updates are of the form $x := C$, which is the basic update action on constraint functions. The update "g is a gender field" followed by "g is male" is a sequence of two basic update actions on constraint functions: $g := Gender; g := \{p_m\}$.

6. Prototype implementation

We have implemented a prototype of the semantics in the lazy, purely functional programming language Haskell [6][1]. The basic type CF which maps a name to a range of values, and the lattice operations are shown in Listing 1.

Actions are functions from constraint functions to constraint functions. The definition of the basic actions skip, update, replace, sequencing, conditional and while loop are shown in Listing 2. Finally, the `info` function, when applied to an action A , gives the constraint function that results from applying A to the state of no information (represented by $\top$/`top`).

We use the following function to encode one enumerated datatype as another one:

```
encode :: (Enum a,Enum b) ⇒ a → b
encode = toEnum ∘ fromEnum
```

[1]The sources can be found here: `http://www.cwi.nl/~jve/software/constraints/CFs.hs`.

```
type Action a = CF a → CF a

skip :: Action a
skip = id

update :: Eq a ⇒ Name → [a] → Action a
update x c f y | x == y    = [ p | p ← f y, elem p c ]
               | otherwise = f y

replace :: Name → [a] → Action a
replace x c f y | x == y    = c
                | otherwise = f y

if_then_else :: (CF a → Bool) → Action a → Action a → Action a
if_then_else condition action1 action2 = λ f →
   if condition f then (action1 f) else (action2 f)

infixl 2 ##
(##) :: Action a → Action a → Action a
a1 ## a2 = a2 ∘ a1

while :: (CF a → Bool) → Action a → Action a
while condition action =
  λf → if condition f
        then (action ## (while condition action)) f
        else f

info :: (Enum a, Bounded a) ⇒ Action a → CF a
info action = action top
```

Figure 2: Actions

Here's how to update with the information that some name x is to be interpreted as a Boolean:

```
isBool :: (Eq a, Enum a) ⇒ Name → Action a
isBool x = update x (map encode (fullRange :: [Bool]))
```

Similar functions can be defined for restricting a field to be of type Char, ASCII, Age, Gender, etc., as long as the basic data type is bounded and can be encoded as Enum. All such "typing" actions follow the pattern of the isBool definition above. Note that it would be possible to write a generic function parameterized on the Bool type, – however, we prefer to keep the examples concrete.

In the same way, partial answers can be modeled as further restrictions of the information space:

```
partialAnswer :: (Eq a, Enum a,Enum b) ⇒ Name → [b] → Action a
partialAnswer x y = update x (map encode y)
```

Finally, a fully answered question is modeled by updating a name to a single value:

```
fullAnswer :: (Eq a, Enum a,Enum b) ⇒ Name → b → Action a
fullAnswer x y = replace x [encode y]
```

Example form instruction:

```
instruction :: (Eq a, Enum a, Bounded a) ⇒ Action a
instruction =
   isAscii  "given name"   ##
   isAscii  "surname"      ##
   isGender "gender"       ##
   isBool   "maried"       ##
   isAge    "age"
```

The form itself can be viewed as the result of updating the state of no information with the instruction:

```
form :: (Eq a, Enum a, Bounded a) ⇒ CF a
form = info instruction
```

Partially answering the form:

```
answer :: (Eq a, Enum a, Bounded a) ⇒ Action a
answer = fullAnswer "given name" (Ascii "Jan")      ##
         fullAnswer "surname" (Ascii "van Eijck")   ##
         fullAnswer "gender" Male                   ##
         partialAnswer "age" [40..70]
```

Finally, here is an example of a conditional statement: initialize the field for "married" on condition that the field for "age" only contains values that lie above 15:

```
marriedQ :: (Eq a, Enum a, Bounded a) ⇒ Action a
marriedQ = if_then_else
               (λf → all (> 15) (map encode (f "age")))
               (isBool "married")
               skip
```

Examples using `while`-loops are more involved since they require user interaction to be meaningful. For instance, if we assume the presence of a "read" function that asks the user for input values of a variable, we could build questionnaires that repeatedly ask for certain questions until the answer is correct or complete.

7. Outlook

This paper makes a small step towards a semantics for questionnaires. It allows us to understand the process of information update when presenting, exploring and answering a questionnaire. Although the current prototype in Haskell provides a nice playground to explore the semantics, it is not suited for direct use in an (embedded) DSL for questionnaires.

Questionnaires based on interactive forms require out-of-order processing of questions. This is handled in the implementation by interpreting forms as constraint functions, where the name fields can be accessed in any order. Still, there is an issue with the purely functional style of the prototype. In interactive questionnaires the user's actions trigger events which result in updates of the questionnaire state. In a purely functional language this can't be directlly expressed. The prototype runs in "batch mode", whereas in an interactive setting, the updates should be incremental. A promising approach would be to combine the model of this paper with functional reactive programming (FRP); see, e.g., [11] for recent advances in this area, or implement the semantics in an impure functional language, such as Scala or F#.

Another opportunity for future work is how to connect this model to state-of-the-art language workbenches, such as Rascal [12, 9, 10]. This would provide full support for defining the (concrete) syntax, compiler and editing support for the language. Initial experiments in formalizing the static aspects of questionnaires are promising[2].

8. Conclusion

Questionnaires are important in society. The use of software for questionnaires is wide-spread, however, they are often still programmed by hand. DSLs could significany improve the quality and effectiveness with which questionnaires are constructed. In this paper we have discussed one aspect of a possible DSL for questionnaires: how can we understand the information flow in questionnaires? It turns out that a model based on epistemic logic can be used to formalize both the definition, presentation and filling out of a questionnaire in one uniform model. We have implemented a prototype of the semantics in Haskell. Future work, however, will focus on how this model can be leveraged in the realization of a complete DSL for questionnaires.

References

[1] Wyatt Allen and Martin Erwig. Surveyor: a DSEL for representing and analyzing strongly typed surveys. In *Haskell'12*, pages 81–90. ACM, 2012.

[2] Blaise homepage. Online. `http://www.blaise.com/`.

[3] P. I. Elsas, Reind P. van de Riet, and J. J. van Leeuwen. Knowledge-based audit support. In *DEXA*, pages 512–518, 1992.

[4] R. Fagin, J.Y. Halpern, Y. Moses, and M.Y. Vardi. *Reasoning about Knowledge.* MIT Press, 1995.

[5] Atul Gawande. *The Checklist Manifesto: How to Get Things Right.* Metropolitan Books, 2009.

[6] The Haskell Team. The Haskell homepage. `http://www.haskell.org`.

[7] J. Hintikka. *Knowledge and Belief: An Introduction to the Logic of the Two Notions.* Cornell University Press, Ithaca N.Y., 1962.

[8] IRS. Forms and publications. Online. `http://apps.irs.gov/app/picklist/list/formsPublications.html` (accessed August 2013).

[9] Paul Klint, Tijs van der Storm, and Jurgen Vinju. EASY meta-programming with Rascal. In *GTTSE III*, volume 6491 of *LNCS*, pages 222–289. Springer, 2011.

[10] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. RASCAL: A domain specific language for source code analysis and manipulation. In *SCAM*, pages 168–177. IEEE, 2009.

[11] Atze van der Ploeg. Monadic functional reactive programming. In *Haskell'13*, 2013. To appear.

[12] Tijs van der Storm. The Rascal language workbench. Technical Report SEN-1111, CWI, 2011.

[13] A. van Deursen, P. Klint, and J. Visser. *Domain-specific Languages*, volume 28, pages 53–68. Marcel Dekker, Inc. New York, 2002.

[14] Centraal Bureau voor de Statistiek (CBS). Brandweerstatistiek 2011: Appendix C, D, E, 2012. `http://www.cbs.nl/nl-NL/menu/themas/veiligheid-recht/publicaties/publicaties/archief/2012/2012-w35-pub.htm` (in Dutch).

[2] For an example questionnaire language (QL), implemented in Rascal, see: `https://github.com/cwi-swat/demoqles`.