Surveyor: A DSEL for Representing and Analyzing Strongly Typed Surveys

Wyatt Allen Martin Erwig
School of EECS
Oregon State University
allenwy@onid.orst.edu erwig@eecs.oregonstate.edu

Abstract

Polls and surveys are increasingly employed to gather information about attitudes and experiences of all kinds of populations and user groups. The ultimate purpose of a survey is to identify trends and relationships that can inform decision makers. To this end, the data gathered by a survey must be appropriately analyzed.

Most of the currently existing tools focus on the user interface aspect of the data collection task, but pay little attention to the structure and type of the collected data, which are usually represented as potentially tag-annotated, but otherwise unstructured, plain text. This makes the task of writing data analysis programs often difficult and error-prone, whereas a typed data representation could support the writing of type-directed data analysis tools that would enjoy the many benefits of static typing.

In this paper we present Surveyor, a DSEL that allows the compositional construction of typed surveys, where the types describe the structure of the data to be collected. A survey can be run to gather typed data, which can then be subjected to analysis tools that are built using Surveyor's typed combinators. Altogether the Surveyor DSEL realizes a strongly typed and type-directed approach to data gathering and analysis.

The implementation of our DSEL is based on GADTs to allow a flexible, yet strongly typed representation of surveys. Moreover, the implementation employs the Scrap-Your-Boilerplate library to facilitate the type-dependent traversal, extraction, and combination of data gathered from surveys.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; G.3.0 [*Mathematics of Computing*]: Probability and Statistics—Contingency table analysis

General Terms Domain-Specific Languages, Generic Programming

Keywords DSEL, GADTs, SYB, Typed data analysis

1. Introduction

Data gathered by polls and surveys provide an important basis for all kinds of decisions in areas such as policy planning, economics, and marketing strategies. The prospect of quickly collecting huge amounts of data from a potentially large number of participants through the Internet has led to a dramatic increase in the use of polls and surveys in recent years.

The specification of a survey is a description of what kind of data is to be collected and also how it is collected. Surveys are composed of individual questions that are often grouped into different sections. This observation indicates that surveys are in principle highly compositional, which makes their construction a nice target for DSLs. It is therefore not surprising to find that quite a few languages and systems have been developed for the specification of surveys. A detailed account of existing approaches will be given in Section 6. Here we just want to point out the general structure and some essential components of any surveying system. This will help us with explaining the contributions of this paper.

- A *description language* S for surveys that defines the questions asked, the kind of responses collected, and potential relationships among questions and answers.
- An *execution mechanism* $E : S \to D$ for surveys that gathers data D based on S and stores it.
- A *query language* $Q : D \to D$ for analyzing and transforming the collected data into statistical data.

The query component is in many cases *not* (considered) a part of a survey system, but we believe that this aspect must not be ignored. Of course, a valid stance that a survey system can take is to deliver the data in some standard format (such as XML) for which standard query languages are already available.

With online survey systems such as SurveyMonkey [14] or similar, a survey can be designed, conducted, and analyzed entirely through a web browser. In this case S is included as a set of HTML forms and interfaces through which the survey designer can construct, arrange and configure a survey. In such tools, after designing a survey, an online version can be produced and tested or distributed to respondents with the click of a few buttons, representing E in this sort of system. Finally, with these web tools, the results can usually be dumped into CSV or XML file formats (D) to be loaded and analyzed by external software. Here, Q is represented by the already-existing, standard query strategies for those formats, such as XQuery [2].

Formlets [3] is a similar tool available in Haskell which offers a way to programmatically and compositionally describe how data is collected through HTML forms and, ultimately, collect well-typed data (D). In this case S is the set of compositional building blocks that Formlets provides which can be brought together to construct the final value. E is the form mechanism in HTML and the Formlets integration with HTTP servers. Finally, since Formlets produces data in the host language, the provisions for Q are merely

Haskell'12, September 13, 2012, Copenhagen, Denmark.

those already available for any data type (such as Haskell's record syntax).

Our approach in Surveyor is to provide *S* as a set of descriptive, strongly typed and composable language features. The features are such that the type (*D*) is evident in *S*. We provide an example of *E* for executing the language on a computer terminal. Finally, we provide a set of generic programming tools that query the data (*Q*) for analysis, taking advantage of type information of the data.

In particular, Surveyor is a *domain-specific embedded language* (DSEL) in Haskell, presenting a way to design surveys that is strongly typed from the ground up. Useful type information is preserved where components of the survey are written, making it easy to use resulting data and automatically verifying correctness. The typed information also makes it simple to describe common analysis tasks within the language.

In Surveyor, a survey is built up of one or more questions. Questions themselves can take different forms. For example, some questions can be answered with any text (free-response questions), while others require one answer to be selected from a finite set of choices (multiple-choice questions). When a survey respondent answers a question, a value is produced which is of a type determined by the type of the question.

The pervasiveness of types in Surveyor makes defining the composition of surveys clear. Since any individual part of a survey has type information associated with it, composing two surveys together will result in a composition of their types. This aids in assembling the parts of a survey because the parts can be written separately and then composed together in the end. It also promotes the reuse of survey code because sets of questions common to surveys can be written once and used in any number of survey scenarios.

Since types are explicitly preserved in Surveyor expressions, analysis tasks can be defined in terms of types as they appear in surveys. This strongly typed notion of analysis also allows us to naturally compose simple analyses into larger analyses of higher dimensionality.

The design and implementation of the Surveyor domain-specific language makes the following main contributions.

(1) It demonstrates how the use of GADTs allows a flexible, compositional approach to the description of user-interaction elements. In the Surveyor language, these elements are simple question-and-response building blocks for construction of surveys and the corresponding response data, but this pattern can potentially be used in other DSLs (for example, for describing the composition of multi-media streams or web pages).

(2) It demonstrates that a strongly typed data representation can be maintained through a series of transformations (survey description, data collection, and data analysis), and that it actually aids the definition of queries and analysis tasks.

The two main parts of the Surveyor DSEL are the survey descriptions and the data analysis combinators.

In the remainder of this paper we will look at survey descriptions in detail in Section 2, show how to execute surveys in Section 3, demonstrate Surveyor's data analysis tools in Section 4, and explain some of the implementation details in Section 5. We finish by discussing related work in Section 6 and drawing conclusions in Section 7.

2. Survey Descriptions

In this section we describe the constructs used to represent surveys. We will introduce basic survey constructions in Section 2.1 and discuss the special case of multiple-choice questions in Section 2.2. In Section 2.3 we introduce syntactic sugar for the DSL in the form of smart constructors, and in Section 2.4 we illustrate the possibility for parameterized surveys that can extend the degree of modularity and reuse.

```
type Prompt = String

data Survey a where
 Respond :: Typeable a => Prompt -> (String->a) -> Survey a
 Choose  :: Typeable a => Prompt -> Choice a -> Survey a
 (:+:)   :: Survey b -> Survey c -> Survey (b,c)
 Group   :: String -> Survey a -> Survey a
```

Figure 1. GADT for representing surveys.

2.1 Basic Survey Construction

At its core, Surveyor provides a parameterized data type `Survey a` to represent a survey which, if conducted, would produce a value of type `a`. This definition allows the type of a survey to be highly suggestive of the survey's nature. For example, two extremely simple surveys might ask participants for only their name or age.

```
type Name = String
type Age = Int

name :: Survey Name
age  :: Survey Age
```

Each survey's parameter type indicates tbe type of data that an execution of that survey will produce. The way in which a survey is constructed is not important for its future use as part of other surveys. This fact ensures a high degree of modularity for survey construction.

Survey combinators allow the construction of complex surveys out of simpler ones. For example, a "person" survey that uses the `name` and `age` to ask for both a person's name and age, can be expressed using the `:+:` constructor for surveys.

```
person :: Survey (Name,Age)
person = name :+: age
```

The `:+:` constructor is part of the `Survey` data type, whose definition is shown in Figure 1. As seen in the type signature, this constructor is able to compose two surveys of different types into a larger survey of both types. `Survey` is defined as a Generalized Algebraic Datatype (GADT) containing constructors which can be used and composed to specify the elements of a survey. A GADT is used here rather than a standard algebraic data type in order that the composition of different constructors corresponds to the structure of the type parameter `a`. The `Survey` type definition is shown below.

The first two constructors of `Survey` specify different types of questions, whereas the remaining two specify survey structure. We will describe the first question constructor `Respond`, and the two structure constructors in the following. Then we defer the description of `Choose` to Section 2.2.

The `Respond` constructor can be used to present a question to which the respondent may reply with text. This text response is parsed into the resulting value of the question. The first parameter to the constructor is the text displayed as the prompt of the question and the second parameter is the "parsing" function which is able to convert the response into an appropriate value of type `a`.

For example, one might use this constructor to produce a question which asks for the respondent's name in the following way. Because `String` is the destination type, `id` is an appropriate parsing function.

```
name :: Survey Name
name = Respond "Your name" id
```

```
data Choice a where
 Item   :: Typeable a => Prompt -> a -> Choice a
 (:|:)  :: Choice a -> Choice a -> Choice a
 (:||:) :: Choice b -> Choice c -> Choice (Either b c)
 (:->:) :: Typeable b => Choice b -> Survey c -> Choice (b,c)
```

Figure 2. GADT for representing Choices.

If the destination type were something other than `String`, then the parsing function would naturally be different. For example, the `age` survey must produce an `Int` value, which can be obtained from entered text using the standard `read` function. [1]

```
age:: Survey Age
age = Respond "Your age" (read :: String -> Int)
```

As mentioned before, the implementation of a survey has no implication on its use since the survey type serves as an interface. For example, we could imagine a more sophisticated function that ensures the `Age` value is within a certain range.

The purpose of the `Group` constructor is to attach a textual heading to a section of a survey. As shown in the type signature, a `Group` is given a `String` to use for a heading and a `Survey a` to enclose. Since the final type of the `Group` constructor is identical to that of the `Survey` it wraps, one can infer that its purpose is strictly presentational—it does not change the survey it wraps. A simple example use of `Group` is shown below.

```
personal :: Survey (Name,Age)
personal = Group "Personal Information" person
```

2.2 Multiple Choice Questions

The `Choose` constructor represents the second type of question that can be included in a survey. In this case, the question specifies a finite set of valid responses which the respondent may select from as an answer. As the type signature shows, the constructor is provided with a textual prompt to display for the question, as well as a `Choice a` expression, which completely determines the type encapsulated by the question (`Survey a`).

From this perspective, `Choose` questions may seem simple, but the choice expression is actually where the power lies and can be fairly intricate. Figure 2 shows the definition of the `Choice` GADT. The use of a GADT again ensures that the structure of the expression matches with the resulting type. We discuss the constructors of the `Choice` type in detail in the following.

The `Item` choice constructor is used to specify one item of a multiple-choice question. The constructor is to be given the prompt for the option as well as the value which is to result, should that choice be selected by a respondent.

The `Item` constructor will typically be used repeatedly in conjunction with the `:|:` constructor that composes two choices. As shown in the type signature, the choices which are composed must be of the same type, and the resulting `Choice` type is the same as the two which are composed. Equipped with this constructor, we are able to create useful multiple-choice questions.

```
rating :: Survey Int
rating = Choose "Rating" $
  Item "Good" 3 :|: Item "Fair" 2 :|: Item "Poor" 1
```

Options of differing types can be composed together in a similar fashion using the `:||:` operator. As is shown in the type signature, a choice of type `b` is composed with a choice of type `c` to form a larger choice of type `Either b c`. The meaning of the `Either` type is that if a respondent were to select one of the options in the left-hand set of choices, the resulting type would not be the same as if they selected something from the right-hand set. But, since the selected option in any case will be one or the other, Haskell's `Either` type is sufficient to unify these two types.

We could use this to create a multiple choice question which results in an `Either` in the following way.

```
voteQuestion :: Survey (Either Char Bool)
voteQuestion = Choose
    "Did you vote/who did you vote for?" $
         (Item "Candidate A" 'a' :|:
          Item "Candidate B" 'b' :|:
          Item "Candidate C" 'c')
    :||: (Item "Didn't vote"    False :|:
          Item "Rather not say" True)
```

Finally, we can create a form of *dependent survey*, a survey that is partially conditioned on specific responses from other parts of a survey. This kind of survey is quite common in larger surveys or questionnaires since they conveniently allow the diversification of the survey and the customization to particular situations of respondents. We want to express this kind of survey by tying a complete survey to a specific answer of another survey question. This is realized by using the `:->:` constructor of the `Choice` GADT. As a simplification for the involved type description we introduce the following type abbreviations `DepSurvey a b` for dependent surveys of type `b` that depend on a value of type `a`, and more specifically, `CondSurvey b` for a survey that depends on a boolean value.

```
type DepSurvey a b = Survey (Either a (a,b))
type CondSurvey b = DepSurvey Bool b
```

As an example of a dependent survey, consider a survey that asks a respondent's dietary restrictions if they have previously indicated that they have any to declare.

```
diet :: CondSurvey String
diet = Choose "Do you have any dietary restrictions" $
            Item "No" False
       :||: Item "Yes" True :->: Respond "Which?" id
```

With dependent surveys one can easily create cascading surveys by repeatedly attaching different surveys to different responses.

2.3 Survey and Choice Smart Constructors

Being a DSL embedded into Haskell brings along the benefit for Surveyor of having access to the expressive computational features of the host language. This allows us, among other things, to keep the core data types for surveys and choices relatively small, but at the same time add syntactic sugar through function definitions. In this section we illustrate how to specialize some of the general-purpose constructors for common use cases by defining a collection of smart constructors.

For example, a common use of the `Respond` question type would be to create a question which simply accepts text. As we've seen above, this is quite easily achieved by using `id` as the parsing function, but we are able to neatly abstract that detail away by providing a smart constructor for the very purpose.

```
text :: Prompt -> Survey String
text p = Respond p id
```

Another common thing is the use of an `Item` prompt as its value. We can define a smart constructor which does just this for both the `Item` and the `:->:` constructor.

[1] It should be noted, the use of the `read` function here introduces a vulnerability to malformed input, and thus, runtime errors. A more complex function could be used which makes use of validation or a default value. Here we use `read` for the sake of simplicity.

```
prompt :: Prompt -> Choice Prompt
prompt p = Item p p

prompted :: Prompt -> Survey a -> Choice (Prompt,a)
prompted p = (prompt p :->:)
```

In a similar way, to construct an option with a type `a` which implements `Show a`, we also provide two smart constructors which can simply use the `String` representation of the `Item`'s value as the text to display. A smart constructor for `:->:` is provided as well.

```
showItem :: (Show v,Typeable v) => v -> Choice v
showItem v = Item (show v) v

vdep :: (Show v,Typeable v) => v -> Survey b -> Choice (v,b)
vdep v = (showItem v :->:)
```

In order to be able to specify the options for a multiple-choice question through a list, we define a smart constructor called `showItems` that accepts a list of `a` values and yields a `Choice` of `a`.

```
showItems :: (Show a,Typeable a) => [a] -> Choice a
showItems = foldr1 (:|:) . map showItem
```

With `showItems` we can construct a short form for encapsulating multiple-choice survey questions as follows.

```
(???) :: (Show a, Typeable a) => Prompt -> [a] -> Survey a
(???) p = Choose p . showItems
```

With the `showItems` smart constructor, one could even build multiple-choice options programmatically. For example, in the following code, the multiple choice question asks the respondent to select a square number, however, instead of hard-coding the options, we can write the numbers with a list comprehension.

```
square :: Survey Int
square = "Pick a square" ??? [ x*x | x <- [1..10]]
```

We can also predefine particular choices that will often occur in multiple-choice questions. Here are two examples.

```
yes :: Choice Bool
yes = Item "Yes" True

no :: Choice Bool
no = Item "No" False
```

With these smart constructors we can redefine, for example, the `diet` survey from the previous subsection in the following, more concise way.

```
diet :: CondSurvey String
diet = Choose "Do you have any dietary restrictions" $
              no :||: yes :->: text "Which?"
```

This definition reveals a new survey pattern, namely that of an *optional survey*, which is a survey only performed if the respondent "agrees" to enter it by answering a lead-in question with yes.

```
(==>) :: Prompt -> Survey a -> CondSurvey a
p ==> s = Choose p $ no :||: yes :->: s
```

With this survey combinator we can finally express `diet` quite elegantly as follows.

```
diet = "Do you have any dietary restrictions"
       ==> text "Which?"
```

2.4 Reusable and Parameterized Surveys

Another useful tool that we can provide to the survey designer is a set of simple, basic surveys representing common sets of questions, which can be reused at will.

Unless a survey is meant to be anonymous, it is very likely that it would ask for a respondent's name. After having identified this common trait of surveys, we are able to provide this functionality to be easily used in any number of surveys.

```
type FullName = (Name,Name)

fullName :: Survey FullName
fullName = text "First name" :+: text "Last name"
```

In this case, `fullName` is, itself, a complete survey, but since Surveyor is nicely compositional, this small example can be incorporated into large survey expressions.

Another commonly found survey question is one that asks for the respondent's gender. In this case, we build something similar to the `fullName` basic survey, except that it uses an algebraic data type in a multiple-choice question.

```
data Gender = Male | Female
     deriving (Eq, Show, Typeable, Enum, Bounded)

gender :: Survey Gender
gender = "Gender" ??? [Male, Female]
```

This pattern of using all values of an enumeration type as values for a multiple-choice question is quite common, and we can support it through a type class `ValueGen` that provides a function to generate all values of a data type automatically.

```
class (Bounded a,Enum a) => ValueGen a where
  values :: [a]
  values = enumFrom minBound
```

With this class we can rewrite the `gender` survey in a slightly simplified way.

```
instance ValueGen Gender

gender :: Survey Gender
gender = "Gender" ??? values
```

The use of `enumFrom` and `minBound` in the definition of `values` explains why we need to derive `Enum` and `Bounded` for `Gender`. We could make this definition even smoother by employing Haskell generics to let the type class `ValueGen` be derived automatically for `Gender`.

It can also be very useful to provide parameterized survey parts. For example, it is common for surveys to ask a respondent to answer a question on a Likert scale (for example, when presented with a statement, the respondent answers with one of Strongly Disagree, Disagree, Neither Agree nor Disagree, Agree, or Strongly Agree). This type of question could be very nicely represented with a multiple-choice question, and we can provide an algebraic data type for the scale to reuse in several of these multiple-choice questions.

```
data LikertScale = StronglyAgree | Agree | NeitherNor
                 | Disagree | StronglyDisagree
                 deriving (Eq, Typeable, Enum, Bounded)

instance Show LikertScale where
  show StronglyDisagree = "Strongly disagree"
  show Disagree         = "Disagree"
  show NeitherNor       = "Neither agree nor disagree"
  show Agree            = "Agree"
  show StronglyAgree    = "Strongly agree"

instance ValueGen LikertScale
```

However, each use of the Likert scale type would have to repeatedly build the choice expression. We can do better. We give this survey

part as a function which accepts a prompt and yields a survey which asks the respondent to answer on the 5-point Likert scale.

```
likert :: Prompt -> Survey LikertScale
likert = (??? values)
```

An example use of this parameterized survey might look like the following. The parameter is a statement with which the respondent may agree or disagree along the scale.

```
feedback :: Survey LikertScale
feedback = likert "My experience was positive"
```

3. Running Surveys

Having an explicit representation for surveys in the form of a data type allows us to separate the questions of survey specification from execution of surveys. Specifically, the execution of a survey can happen on any number of platforms, and the presentation of a survey could be rendered to any number of formats. For example, after having written a single survey in Surveyor, one might decide to render it as an HTML web page, or to print it out on paper, or to do both. The structure of the Surveyor language is decoupled from the destination medium so that the same survey could be rendered to any of them. Our main motivation for designing our DSL was to identify the essential structure of surveys and their relationships to collected data. Details regarding the user-interaction with a survey or engineering aspects related to questions of survey distribution or the storage of data are important, but not our main concern here.

Therefore, for the examples considered here, we describe how Surveyor provides a function for rendering a survey as an executable program with a Command-Line Interface (CLI). As shown in the method signature below, the `runSurvey` function takes a `Survey` which results in a value of type `a`, and produces an operation in the `IO` monad which produces a value of type `a`. This operation is the CLI which poses questions and collects responses.

```
runSurvey :: Survey a -> IO a
```

This function serves as a design and debugging aid for the survey designer, making it easy to test survey expressions by executing them, actually answering the questions and observing the resulting value.

To illustrate the execution of surveys with a small example, we imagine a survey designer in the sales department of a company that produces spiral-bound notebooks. The designer decides to build and conduct a survey to find out what customers think of the notebooks.

To start with, our designer builds a small survey to ask basic questions and get context. In this example, `Handedness` is an algebraic data type to represent left- or right-handedness and is implemented in the same way as the `Gender` data type.

```
data Handedness = LeftHanded | RightHanded
     deriving (Eq, Show, Typeable, Bunded, Enum)

instance Value Handedness

handedness :: Survey Handedness
handedness = "Handedness" ??? values
```

With this definition the designer can create a basic survey about the user's background.

```
type Personal = ((FullName,Gender),Handedness)

background :: Survey Personal
background = name :+: gender :+: handedness
```

Next, the designer needs to write the part of the survey which collects the customer sentiment. For simplicity's sake, in this case, we ask whether the respondent owns one of the notebooks, and, if they do, a Likert question is used directly asking what they think of it.

```
sentiment :: CondSurvey LikertScale
sentiment =
   Choose "Have you bought one of our notebooks?" $
      no :||: yes :->: likert "The notebook is good"
```

Finally, the designer can compose these two together into the final survey.

```
type Notebook = (Personal,CondSurvey LikertScale)

notebook :: Survey Notebook
notebook = background :+: sentiment
```

We can run this survey now using the function `runSurvey`, which then produces the following output (for brevity we show only the portion for the personal data). Note that the listing includes both the output from the survey and the input from the participant.

```
*Main> runSurvey $ Group "Personal Data" background

Personal Data
=============

First name: John

Last name: Smith

Gender:
[1] Male
[2] Female
1

Handedness:
[1] Left handed
[2] Right handed
2
((("John","Smith"),Male),RightHanded)
```

As shown above, the title attached to the survey with the `Group` constructor is presented in this interface as a heading which precedes the questions it groups. The question responses of `John`, `Smith`, `1` and `2` are typed by the respondent where the question is presented and must be entered before the interface moves on to the next question. When the survey is complete and all the questions have been answered, the resulting value is displayed by the environment as the result of the `IO` operation.

If this survey were to be conducted and several people responded to it, then a set of the produced values might look something like this.

```
responses :: [Notebook]
responses = [
 ((((("John","Smith"),Male),RightHanded),
    Right (True,Agree)),
 ((((("Jane","Doe"),Female),LeftHanded),Left False),
 ...
 ]
```

4. Typed Data Analysis

Having explored the building blocks of a survey, and seen an example of building a survey that has some usefulness, the remaining challenge is making use of respondent data with analyses of varying dimensionality.

Imagine having accumulated a set of data such as the responses to the customer survey in the previous example (`responses`). If we want to understand this dataset, two problems immediately present

themselves. First, the actual type of the data is a somewhat deeply nested tuple structure which may be difficult to read by simply printing. Second, the survey (ideally) will have received a large number of responses from customers, and there will be too many to make any significant determination from them without the aid of a tool. These problems can be solved with the analysis parts of the Surveyor DSL.

We can categorize analyses by their *dimensionality*, by which we mean the number of different types in the data that an analysis focuses on.

We will present several examples of 0-, 1-, and 2-dimensional data analyses in the following subsections.

4.1 0-Dimensional Analyses

A 0-dimensional analysis of data ignores any specific type and produces a single value, typically through some form of data aggregation. A trivial example would be to count the number of responses using Haskell's `length` function. As is indicated by the type of `length` this analysis is completely generic because it does not consider any of the values that are contained in the data. Even such a trivial analysis can be useful because we can use it to make a determination of the data, such as whether it is in sufficient number to bear statistical significance.

Other 0-dimensional analyses could compute a maximum or minimum of some sort, which will depend on types of the survey data, but not on a specific one.

4.2 1-Dimensional Analyses

In the notebook customer-survey data, it might be useful to see whether the survey was responded to by more women or more men. This is an example of 1-dimensional analysis because it examines how one part of the survey was answered. Indeed, because Surveyor is strongly typed, we can more precisely say that 1-dimensional analysis, in this case, is examining the values of *one data type* as it appears in the survey.

However, before we can analyze the one particular aspect of the survey, we need a systematic way of extracting it from any of the responses. Surveyor provides two functions (called accessors) which aid in this task.

The first accessor tool is a generic function which, given the actual specification of the survey involved (of type `Survey a`), is able to retrieve a value of some arbitrary destination type (`b`) from an answer to the survey (a value of type `a`).

```
guidedBy :: Typeable b => Survey a -> a -> Maybe b
```

The result of running the accessor is `Maybe b` rather than simply `b` to account for the possibility that a value of the target type is not present in the particular answer (for example, if it would only be present in the untaken branch of an `Either` construct, or, indeed if the survey did not involve the target type at all). We could use this function to get a list of all the `Gender`s that were provided as answers in the survey data.

In the below example, we show the use of this accessor for extracting a distribution of `Gender`s from the data. In doing so we employ an auxiliary function `scan` that uses the `catMaybes` function from Haskell's `Data.Maybe` module to filter out any `Nothing` values produced by the accessor.

```
scan :: (a -> Maybe b) -> [a] -> [b]
scan f = catMaybes . map f
```

Since `guidedBy` is a generic function, its return type is determined by Haskell to be `Maybe Gender` from the type signature of the `genders` declaration.

```
genders:: [Gender]
genders = guidedBy notebook `scan` responses
```

The second type of accessor is a variation of the first and is designed to find a question with a given text as its prompt. This can be useful for extraction of only one of several questions which have the same type in the survey, but different prompts.

```
searchingFor :: Typeable b =>
                String -> Survey a -> a -> Maybe b
```

This function can be used in the same way as `guidedBy`, except that it needs the extra parameter for the question's prompt text.

```
firstNames :: [String]
firstNames = searchingFor "First name" notebook
             `scan` responses
```

It is important not to confuse this use of `searchingFor` to be *indexing* the data by a string. Rather, it searches for answers which meet the string constraint *in addition* to the type constraint.

Finally, equipped with these data accessor tools, we can perform interesting analysis on individual data types. For example, we may wish to see how the two `Gender` answers are distributed throughout the data. We can take advantage of Surveyor's facilities for value distributions to do this. In Surveyor, we have a parameterized type to describe this sort of information called `Dist`.

```
data Dist a b
```

The parameters of the `Dist` type illustrate what information is being described by it. The parameter `a` refers to the type of the survey being analyzed. In our case `a` would be `Notebook`. The second parameter is the type of the values whose distribution is being analyzed. For example, to look at the `Gender` distribution in we might use the following type.

```
genderDist :: Dist Notebook Gender
```

To build a value of this distribution type, Surveyor provides the `collate` function, which, given an accessor and a set of responses, produces the desired distribution.

```
collate :: Eq b => (a -> Maybe b) -> [a] -> Dist a b
```

Using `collate`, the gender distribution can be apprehended with the familiar building blocks. As before, the type of the `guidedBy` is determined by the type signature of `genderDist`, which was given above.

```
genderDist = guidedBy notebook `collate` responses
```

The construction of this distribution has already accomplished an analysis task. To illustrate this we can take advantage of how `Dist` is an instance of the `Show` type class and see what it can say about how genders are distributed throughout the survey.

```
> genderDist
Male:   41.666668%
Female: 58.333332%
```

The above result, which is produced by calling `show` on the distribution, demonstrates at a glance that there were slightly more women than men who responded to the customer survey. Using an identical approach, we can construct distributions for the `Handedness` and the `LikertScale` types, which we call `handednessDist` and `likertDist`, respectively. The following two results are produced. First, for handedness.

```
> handednessDist
RightHanded: 75.0%
LeftHanded:  25.0%
```

And then for the Likert scale we obtain the following distribution.

```
> likertDist
Agree:                        33.333332%
Disagree:                     16.666666%
Strongly agree:               16.666666%
Strongly disagree:            8.333333%
N/A:                          8.333333%
Neither agree nor disagree:   16.666666%
```

The distribution entries listed under `N/A` are from those answers for which the accessor produced `Nothing`, rather than an actual value, meaning, in this case, that the respondent had indicated that they had not bought a notebook and therefore did not rate it.

The `Handedness` distribution shows that there were three times as many right-handed respondents as left-handed ones, while the `LikertScale` distribution shows that opinions of the notebooks are generally positive, which is good news for our imaginary sales department.

4.3 Multivariate Analysis

Surveyor's 1-dimensional value distribution analysis is useful to identify simple trends, but the real goal of analysis is the discovery of *relationships* between multiple parts of a survey. In this strongly typed context, we can state this more precisely as the discovery of relationships between the *values under multiple types* in the survey. This can be described as multivariate analysis, or, as we implement it in Surveyor, 2-dimensional analysis.

Following the principle of compositionality, 2-dimensional analysis can be defined in terms of the composition of two 1-dimensional analyses (of type `Dist`). The combinator provided in Surveyor for this purpose computes a cross-tabulation of the distributed data (also sometimes called a contingency table).

```
crosstab :: Eq a => Dist a b -> Dist a c -> Table b c
```

As is shown in the type signature of `crosstab`, given two distributions that examine values of potentially different types over the same survey type, a cross-tabulation is constructed which lists values of the first distribution type along one axis, and those of the second along the perpendicular axis.

For example, we could use this combinator to attempt to discover any relationship between the gender of the respondents and their dominant hand by composing these two distributions under `crosstab`.

```
genderHandedness :: Table Gender Handedness
genderHandedness = genderDist `crosstab` handednessDist
```

As with distributions, the type `Table` is an instance of the `Show` type class, and we can use this to see what the table can say about these two dimensions.

```
> genderHandedness

        RightHanded  LeftHanded
Male    4            1
Female  5            2
```

There appears to be no outstanding relationship between these two dimensions, and this is just as we would expect. We can continue our hunt by using an identical approach with the `Handedness` and `LikertScale` distributions to check if there is any relationship to be seen between a customer's opinion of the notebook and their dominant hand.

```
> likertDist `crosstab` handednessDist

                              RightHanded  LeftHanded
Agree                         3            1
Disagree                      1            1
Strongly agree                2            0
Strongly disagree             0            1
N/A                           1            0
Neither agree nor disagree    2            0
```

5. Implementation

Here we examine how some of Surveyor's functionality is implemented. As we've seen, the bulk of the language constructs are implemented as GADT constructors. In general, the advantage afforded the Haskell programmer who makes use of GADTs is that, when writing functions, pattern-matching against constructors causes type-refinement—knowing information about the constructor automatically provides type information [7]. In the implementation of Surveyor's functions, we will be making significant use of pattern matching, and this advantage will be evident.

5.1 Run Survey

We look at parts of the implementation of the `runSurvey` function, which, as seen above, is of type `Survey a -> IO a`, meaning that it transforms a survey of some type into a Command-Line Interface for the survey as an IO action, which can produce a value of that type.

The simplest pattern for this function to handle is the `Group` constructor. Since the purpose of this part of the language is merely to attach a label to a section of survey questions, the implementation here will simply print the prompt to the terminal (with a small horizontal rule) and recurse on the enclosed questions.

```
runSurvey (Group name sub) = do
    putStrLn $ "\n" ++ name
    putStrLn $ replicate (length name) '='
    runSurvey sub
```

Another easy pattern for the `runSurvey` function is the survey composition operator. Recall that the type of the survey composition operator is `Survey b -> Survey c -> Survey (b,c)`, so the result of this `IO` operation needs to be the pair of the results of the two operands. This is as simple as recursing on each argument separately, and then packaging the results in a pair.

```
runSurvey (left :+: right) = do
    l <- runSurvey left
    r <- runSurvey right
    return (l,r)
```

Next we look at the implementation for `Respond`. In this code the constructor is identified and the prompt and parsing function are matched to names. The prompt is displayed for the user to respond to and a line of text is read from the standard input. Recall however that if a `Respond` question is a `Survey a`, then the parsing function will be of type `String -> a`. Therefore, in order for this `IO` action to result in a value of type `a`, it must apply `parser` to the line of text.

```
runSurvey (Respond prompt parser) = do
    putStr $ "\n" ++ prompt ++ " "
    ans <- getLine
    return $ parser ans
```

The implementation for `Choose` is slightly more involved. We display the prompt in the same way as we do in the case of `Respond`, and then list the valid answers by calling `dispChoices`. This lists the options on several lines with indices next to each. These indices start with the integer which is passed as the first parameter, so these

will be displayed starting with 1 (one-offset). We read an integer from the standard input and then use it to select an option for use as the final result. This is done by calling the `selected` function with the predecessor of the user-provided index (converting one-offset to zero-offset).

```
runSurvey (Choose prompt choiceExp) = do
    putStrLn $ "\n" ++ prompt ++ ": "
    dispChoices 1 choiceExp
    ans <- readLn :: IO Int
    selected (pred ans) choiceExp
```

The `dispChoices` function is implemented as follows. This function builds an `IO` action which results in an `Int` that represents the successor of the highest index under which an option was printed. Since homogeneous and heterogeneous choice composition are displayed the same way, their display code is wrapped in another helper function called `disp2`.

```
dispChoices :: Int -> Choice a -> IO Int
dispChoices num (Item text _) = do
    putStrLn $ "[" ++ show num ++ "] " ++ text
    return $ succ num
dispChoices num (choice :->: _) =
    dispChoices num choice
dispChoices num (l :|: r) = disp2 num l r
dispChoices num (l :||: r) = disp2 num l r

disp2 :: Int -> Choice a -> Choice b -> IO Int
disp2 l r = do
    next <- dispChoices num l
    dispChoices next r
```

The `selected` function takes the selected index and a choice expression and will yield the value corresponding to that choice (calling `runSurvey` on a dependent survey where necessary). The only reason this function is in the `IO` monad is because it could potentially have to execute `runSurvey`. This follows a simple recursive pattern which works in much the same way as a binary search algorithm.

```
selected :: Int -> Choice a -> IO a
```

When presented with the homogeneous composition of choices, it measures the number of items in the first one. If the index is less than that number, it will recurse with that set of choices, but if it's greater, it will recurse on the second list with an index that is less the number of items in the first. When presented with a heterogeneous composition, the case behaves in a similar fashion, except that it must apply the `Left` and `Right` constructors of the `Either` data type because the choice sets will be of different types. This is done with the `<$>` operator from the `Control.Applicative` module.

```
selected n (l :|: r)
    | n < m     = selected n l
    | otherwise = selected (n-m) r
      where m = choiceLength l
selected n (l :||: r)
    | n < m     = Left  <$> selected n l
    | otherwise = Right <$> selected (n-m) r
      where m = choiceLength l
```

When presented with a dependent choice, the `selected` function must call itself recursively as well as the `runSurvey` function. Recall that the type of a dependent choice is `Choice b -> Survey c -> Choice (b,c)`. Therefore, it first recurses into the choice that is the first parameter of the construction, and must also execute the sub-survey that is the second parameter. The results are paired together according to the type of dependent choices.

```
selected n (choice :->: survey) = do
    c <- selected n choice
    s <- runSurvey survey
    return (c,s)
```

Finally, when presented with an item, the effective size of the choice set in this invocation is zero, so this case is only defined for when the index parameter is zero.

```
selected 0 (Item _ val) = return val
```

To measure the length of choices, the `choiceLength` function is provided, the implementation of which is trivial. Its signature is given below.

```
choiceLength :: Choice a -> Int
```

5.2 Data Analysis

The analysis task in Surveyor is made possible by the generic programming facilities provided by the Scrap-Your-Boilerplate library for Haskell (SYB) [11]. In overall design, the analysis parts of Surveyor abstract generic programming into the accessor functions. The central generic function in Surveyor is `guidedBy`, which was introduced earlier. Its type is repeated here for convenience.

```
guidedBy :: Typeable b => Survey a -> a -> Maybe b
```

When given a survey of some type `a`, and when called with target type of `b` (which has to implement the `Typeable` type class), it is able to yield a function which, given a survey value, will yield a `Maybe b`. That is, `guidedBy` will try to find the value of type `b` within a value a of type `a`, but if it cannot succeed, it will return `Nothing`.

To handle survey composition, `guidedBy` recurses twice and coalesces the resulting values with Haskell's `orElse` function. If the accessor succeeds in either branch, the value will be used, with preference going to the left.

```
guidedBy (left :+: right) = \x -> orElse
    (guidedBy left  $ fst x)
    (guidedBy right $ snd x)
```

When the `guidedBy` function is called with `Respond`, the problem is reduced to a use of the `cast` function provided by SYB. This function constitutes a type-safe cast tool given by the type `(Typeable b, Typeable a) => a -> Maybe b`, which will convert types where it can and deliver the result within a `Just` constructor; however, if it cannot, it will yield a `Nothing`. This is precisely the behavior we want for a value resulting from a `Respond` question.

```
guidedBy (Respond _ _) = cast
```

The case for `Group` constructions is trivial: the accessor recurses directly on the sub-survey. In the case for handling `Choose` questions, the `genericChoiceAccessor` function is used, which is implemented with the same strategy as the `guidedBy` function, except that it is designed to traverse across choice expressions rather than survey expressions.

```
guidedBy (Group _ s)  = guidedBy s
guidedBy (Choose _ c) = guidedByChoice c

guidedByChoice :: Typeable b => Choice a -> a -> Maybe b
```

The `guidedBy` function suffers from the limitation that it cannot differentiate between values of the same type. For example, imagine a survey which involves two `likert` questions. Since `guidedBy` is built with a left-preference, it would only be able to access the first one. To solve this problem, we also provide the function `searchingFor`, which is built on top of `guidedBy` to be able to differentiate questions of the same type.

The `searchingFor` function operates with the same traversal pattern as `guidedBy`, except that it searches with the condition of matching a `String` value against the `Prompt` value of questions. Where it finds a match, it immediately defers to the `guidedBy` function to actually do the casting. In a similar way, the remainder of the analysis functionality provided in Surveyor is built upon the single generic building block of the `guidedBy` function and do not themselves need to directly make use of the generic programming tools from SYB.

As we saw in section 4.2, the `Dist` type can be used as the result of 1-dimensional analysis and the type is parameterized by the type of the survey and by the type under analysis. The complete type definition is below. It shows that, under the covers, a distribution is a list of pairs. This list act likes a dictionary, with values of the type `Maybe b` as the keys and lists of the full survey answers as the values.

```
data Dist a b = Dist [(Maybe b, [a])]
```

The reason that `Dist` uses this representation is so that it can track the unique values of type `b` from the survey as well as know which answers correspond to them. This is enough information to compute the percentage-based distribution analysis that a distribution presents when passed into `show`. It counts the total number of answers represented in the distribution and, for each value in the type `b`, it finds the proportion of the total answers which had that value. The key for this list must be wrapped in `Maybe` to be able to still account for those times that the accessor fails to withdraw a value. To produce a distribution with the `collate` function, the `Eq` typeclass is needed for the type `b` in order to compare the values extracted from answers, but, beyond that, the implementation is straightforward.

6. Related Work

There is no shortage of survey tools in the wild. Many of these tools are online, including LimeSurvey [12], SurveyMonkey [14], and SurveyGizmo [13], as well as ad-hoc polling functionality in social networks, such as Facebook Questions. An important advantage to online survey systems is the way that the entire survey lifecycle can take place in the same online space. Some of these tools even provide simple (1-dimensional) analytic tasks on the survey data, but more involved analyses can only be accomplished by leaving the online environment and importing the data into a specialized analysis tool such as IBM SPSS [9].

SPSS itself is an important tool which is able to encompass the entire lifecycle of data collection and analysis. With SPSS, however, the emphasis is placed on analysis tasks, and the survey component is essentially a more business-oriented version of the survey tools provided with online survey systems. The analysis capabilities in SPSS are highly sophisticated, but are designed to be generally applicable to any dataset. In particular, they are not designed specifically to analyze surveys. In a similar way, the R programming language [10] provides tools for sophisticated analysis on arbitrary data.

Hage and van Keeken's Neon DSL [8] represents a related approach to constructing and composing analytic tasks as typed functions in the Haskell host language. After having collected a large, somewhat cumbersome dataset of compilation problems encountered by students working on programming assignments, they implemented Neon as a tool to easily and functionally explore the dataset and draw conclusions about language use and error fixing. Moreover, Neon's design is motivated by the application of descriptive statistics (to summarize the large dataset, and draw general conclusions on trends in language use), whereas our analysis tools in Surveyor are motivated by inferential statistics (to elicit relationships within the data).

In our design, we drew upon the inspiration of Cunningham's "Little Language for Surveys" [4]. Although its purpose was mainly to demonstrate how one might implement a DSEL in the Ruby programming language, the example illustrates how the problem of survey specification is amenable to a domain-specific treatment. In this Ruby DSEL, one can construct surveys made up of multiple-choice questions that behave in very much the same way that they do in Surveyor. The main difference is that the Ruby DSEL creates surveys that carry no explicit type information, and thus the data is dynamically typed (as is expected in Ruby programming). Data is collected by the execution of Ruby block functions against a common scope, and the blocks are programmed to destructively assign values in variables to store survey respondent data. Furthermore, conditional parts of a survey are determined to be included or excluded based on the result of executing Ruby blocks that test against variables in this common scope. As a consequence, these Ruby surveys are neither strongly typed nor truly compositional. The composition of two surveys could potentially involve a name conflict resulting in the loss of respondent data and even potentially an unanticipated change in the type of the data. With Surveyor, we instead explicitly define how survey composition results in the composition of survey types. This makes conflicts between survey parts impossible, ensuring data cannot be lost and making the resulting types of data completely predictable.

Cunningham himself calls upon inspiration from the "Little Languages" column by John Bentley [1], in which Bentley makes several cases for the benefits of DSLs. He also presents an example of a DSL for surveys. Like the Ruby DSL, this language only involves multiple choice questions and provides a way to make some of them conditional in terms of other answers. However, it does not resort to scope trickery to store answers, but questions must explicitly state database column indices in which their answers are to be stored. This prevents the language from being compositional in a similar way as the Ruby DSEL: conflicts between column indices are possible.

The systematic collection of strongly typed data as propagated by Surveyor is not new. Formlets [3] provide an excellent example of populating typed data structures by compositionally assembling lenses that compile to an HTML form as the interface. Our approach differs from Formlets in two important ways.

(1) In Surveyor, the type of the survey is defined by the structure of the survey. This is a natural way to have the data typed because the specification of a survey is basically a description of how data is to be collected. From this the type of the data to be collected follows naturally. Compare this approach with Formlets, where the type of the data is chosen beforehand, and the specification is a technique for populating the type.

(2) In Surveyor, the final representation is not bound to any specific format. A Surveyor expression could be compiled into HTML, but could just as easily be run on a text terminal (as we've seen) or be rendered to a print document. Contrast this with Formlets, where the language is specifically geared towards HTML. Even though in principle, Formlets could target other representations than HTML, it doesn't make a clear, purposeful decision to separate the description from this design commitment.

Our approach to type composition was partially inspired by FunctionalForms [6], which is a Haskell library for strongly typed GUI dialogs. In FunctionalForms, a dialog is a data type parameterized by the type of values it produces when executed. Composition of these dialogs results in composition of the result type in much the same way as this occurs in Surveyor. However, FunctionalForms takes extra measures to minimize the complexity of the resulting type. In our approach, we allow a highly composed Survey type to

become deeply nested, but mitigate the difficulty of dealing with such a type by also providing easy-to-use analysis tools.

7. Conclusions & Future Work

We have introduced the Surveyor DSL for constructing strongly typed surveys. The implementation as a DSEL in Haskell relies crucially on the concept of GADTs and the type-directed generic programming facilitated by the Scrap-Your-Boilerplate approach. The incorporation of types into the representation of surveys has provided two distinctive benefits. First, the data collected using typed surveys will also be typed and can thus be analyzed in a more flexible, yet still type-safe way. Second, we found that the need to treat the types of surveys systematically in compositions was a helpful guide in actually designing the final representation and thus the DSL.

In future work, we plan to evaluate Surveyor by using it as part of ongoing end-user research at Oregon State University [5]. To this end we are currently working on completing a web interface for the Surveyor language that is entirely based on HTML and CCS and independent from other tools (such as the jQuery library or Apache). We will use this interface to conduct real-world surveys and analyze the gathered data.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful and constructive comments.

The first author wants to thank Eric Walkingshaw for his help with type-directed algorithms.

References

[1] J. L. Bentley. Programming Pearls: Little Languages. *Communications of the ACM*, 29(8):711–721, 1986.

[2] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon, editors. *XQuery 1.0: An XML Query Language (2nd Edition)*, 2010. `www.w3.org/TR/xquery/`.

[3] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. An idioms guide to formlets. Technical Report EDI-INF-RR-1263, University of Edinburgh, 2008.

[4] C. H. Cunningham. A little language for surveys: Constructing an internal DSL in Ruby. In *Proceedings of the ACM SouthEast Conference*. ACM Press, 2008.

[5] EUSES. End users shaping effective software. `http://EUSESconsortium.org`.

[6] Evers, S., Achten, P., Kuper, J. A functional programming technique for forms in graphical user interfaces. . In *16th International Workshop on Implementation and Application of Functional Languages*, 2004.

[7] The glorious glasgow haskell compilation system user's guide. `http://www.haskell.org/ghc/docs/6.6/html/users_guide/gadt.html` (Last accessed: June 2012).

[8] J. Hage and P. van Keeken. Neon: a Library for Language Usage Analysis. In *Int. Conf. on Software Language Engineering*, pages 33–53, 2008.

[9] IBM SPSS Software. `http://www-01.ibm.com/software/analytics/spss/` (Last accessed: June 2012).

[10] R. Ihaka and R. Gentleman. R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

[11] R. Lämmel and S. Peyton Jones. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 26–37, 2003.

[12] LimeSurvey. `http://www.limesurvey.org/` (Last accessed: June 2012).

[13] SurveyGizmo. `http://www.surveygizmo.com/` (Last accessed: June 2012).

[14] SurveyMonkey. `http://www.surveymonkey.com/MySurveys.aspx` (Last accessed: June 2012).