Topsl: A domain-specific language for on-line surveys

Mike MacHenry
Northeastern University
dskippy@ccs.neu.edu

Jacob Matthews
University of Chicago
jacobm@cs.uchicago.edu

Abstract

There are currently few choices for social scientists who want to employ web-based surveys in their studies. They can either use a special-purpose language whose notion of flow control may be too limiting to express advanced survey designs, or use a general-purpose language that gives them the freedom to make complicated survey designs but makes them reimplement infrastructure code for saving questions to disk, generating HTML, and so on with each new survey. In this paper, we introduce Topsl, a domain-specific language embedded in PLT Scheme that takes the middle road, giving survey authors a way to reuse survey infrastructure for new surveys while also allowing them to express complicated survey designs easily.

1 Introduction

As social scientists have become more aware of the practical and theoretical benefits of gathering information online [2], the demand for web-based surveys has grown significantly in recent years. Unfortunately, technology has not improved to meet this demand. Social scientists want to design surveys that interact with participants in complicated ways that current survey languages are not capable of expressing.

Existing domain-specific languages (DSLs) for on-line surveys such as SuML [1] and QPL [10] have a limited notion of control flow. In all domain-specific survey languages the authors have found, the fundamental notion of the flow from one piece of a survey to the next is built-in and unchangeable. That means that while simple surveys are easy to implement, if a programmer wants to add a seemingly minor extension that affects the survey's flow, he or she may find the task impossible.

For example, the authors were introduced to this problem by Dr. Eli Finkel, assistant professor of psychology at Northwestern University, in the fall of 2003. Finkel found that while a plethora of contracting companies thought it would be technically feasible to loop over a user-provided input list if the user provided all items at once, none could provide the same looping facility if the user provided the items one at a time over multiple survey sessions.[1]

When studies run into these limitations, programmers resort to implementing them in a general-purpose language (GPL) such as PHP or Perl that allow them to express anything they want (as evidence that this is a popular approach, Fraley recently published a how-to guide on the subject for psychologists [4]). Unfortunately, if they make that choice, they become responsible for handling HTML generation, CGI, and data storage, all of which is unrelated to the specific survey being written. In the authors' direct experience, on-line surveys are plagued by bugs in this non-domain-specific code. For instance, in one case an on-line sociology survey implemented from scratch had a bug in its answer-saving routines that caused it to lose a significant portion of answers. When the bug was discovered, the researchers had to contact all the participants and ask them to fill the survey out again; only a fraction of the participants actually did. Such incidents, though common, are an unacceptable risk in expensive research.

It is natural that these two general strategies for solving the survey problem should emerge. Survey programs exist to collect answers to questions that will then be put into rows in a database or analyzed by a statistics program, and that might be printed out for copy-editing or for handing out to off-line survey participants. To make those operations possible, all the questions a particular survey could ask must be statically identifiable. Of course if a survey program had complete freedom at runtime to generate questions, that identification would be impossible. So, the problem must be made easier, and two simple ways to make it easier are to restrict the language in which programmers write surveys to the point where questions are statically identifiable, or restrict analysis to one particular survey and do the analysis by hand.

Both available options have serious problems, though: current DSLs afford too little flexibility in their models of flow control, and GPLs make programmers implement substantial amounts of non-domain-specific code for each survey. In this paper, we demonstrate a way to take the middle path with Topsl, a domain-specific language for writing on-line surveys embedded into the general-purpose language PLT Scheme. We arrange the embedding so that programmers can write survey code without having to worry about non-survey-specific concerns, but can use the full power of PLT Scheme when it becomes necessary.

We begin by explaining our design goals for the language, then

[1] Since survey authoring companies use similar in-house survey creation software they suffer from the same limitations.

Fifth Workshop on Scheme and Functional Programming. September 22, 2004, Snowbird, Utah, USA.

present the language's syntax and semantics, and then discuss its implementation. Afterwards we discuss how the unique features of Scheme and PLT Scheme proved useful to our implementation, assess our results and point out some directions for future work.

2 Design Goals

Topsl's primary design goal is to provide a layer of abstraction that allows programmers to express surveys clearly and without having to write non-domain-specific code while still being expressive enough to write surveys with novel control-flow elements. Further, it should be easy-to-use in the sense that valid programs cannot fail to save data or otherwise malfunction and a program's final product, a series of answers, should be immediately ready for analysis in regular statistics programs.

2.1 Interoperability

One major motivation for Topsl is to eliminate the need for a programmer to write code that connects the fundamentally novel aspects of a particular survey to the technologies that manage presenting that survey to participants and saving their responses. To that end, the language must make that management invisible and unbreakable; it must not require any special action on the programmer's part for a survey to appropriately interact with participants and save results.

Topsl also needs to interface with existing technology used by social scientists in order to be useful. When social scientists conduct surveys, they store results in statistics program databases to analyze results. Importing results into a statistics program requires prior knowledge of the questions since a column will need to be created in the database for each possible question. So when social scientists design on-line surveys, they need the systems they use to create static summaries of their surveys, containing a set of all possible questions. Each question needs an identifier consistent with the dynamic implementation and question text with possible place holders where dynamic information will be filled in at runtime. Topsl must ensure that all surveys can be analyzed to produce a static summary.

2.2 Expressiveness

Handling web-programming details and interfacing with statistics programs is not enough. Social science surveys commonly use complex designs with control flow such as that present in Finkel's survey. For instance, the authors recently heard of a novel survey design in which husband and wife pairs participated cooperatively. Within each pair, one participant's answers affected the questions the other was asked. Topsl should allow programmers to express such complicated calculations using full Scheme where necessary.

In his paper, "A Universal Scripting Framework" [11], Shivers advocates the embedding of domain-specific languages within general-purpose languages to increase the expressiveness of the DSL. Topsl needs to allow programmers to "break out of the [DSL] in order to express complex computations in a [GPL]" so that complicate surveys are possible to write.

Unfortunately, allowing full Scheme in Topsl programs is at odds with our goal of automatically generating static summaries. Allowing the programmer to generate surveys using arbitrary Scheme makes statically listing all possible questions undecidable for many surveys, so some compromise is inevitable. However, since in practice even surveys with exotic designs have an obvious and predictable set of questions that will be asked, Topsl can realistically enforce a restriction that all surveys written in the language be analyzable to produce a static summary while still letting programmers write surveys with complicated control flow.

2.3 Language Growth

Languages that try to guess up-front everything their users will ever want to do tend to find that they've guessed wrong. In "Growing a Language," Steele writes that "...a main goal in designing a language should be to plan for growth" [7]. Topsl programmers should be able extend the language with new features to meet these needs. Just as in Scheme, Topsl users should be able to extend Topsl's syntax and add new functions as necessary — in addition, they should be able to add new question types and formatting options if need be. Topsl must be carefully designed to let programmers create these extensions in such a way that surveys using them remain statically analyzable.

3 Design

A Topsl survey contains a definitions context and any number of Topsl forms which make up the survey's control flow. The most basic Topsl form is the page, which is constructed from any number of page elements. The most basic page element is the question (written **?**). Questions take a question type and any number of strings or variables that make up the question text. Topsl provides basic question types like *free* (for free response questions) and *yes-no* as well as functions that produce new question types like *radio* and *multi-select* for selecting from a list of responses. A simplified Topsl syntax is as follows:

survey	::=	<definition>* <survey-element>*
definition	::=	(**define** <variable> <scheme>)
survey-element	::=	(**page** <page-element>*)
page-element	::=	(**?** <question-type> <question-text>*)
question-type	::=	<variable>
question-text	::=	<string> | <variable>

We present Topsl as a series of example surveys, augmenting the syntax with new forms as examples become more complicated. We start with a trivial example survey and show how to add abstraction and control flow to the language while still retaining the ability to generate a static summary.

3.1 A Simple Survey

We can use **page** and **?** along with a few question types to create a simple example of a complete Topsl program. In the following example, we define a new question type, *enjoy*, and then create two pages containing two questions each.

```
(define enjoy (radio "A lot" "Some" "Not at all"))
(page (? free "Where did you go to high school?")
      (? enjoy "How did you like it?"))
(page (? free "Where did you go to college?")
      (? enjoy "How did you like it?"))
```

When a participant visits the web page associated with this survey, the survey will display a page containing the appropriate questions (see figure 1). When the participant fills out the form and submits it, a second similar page will be presented, followed by a page indicating that the survey is over.

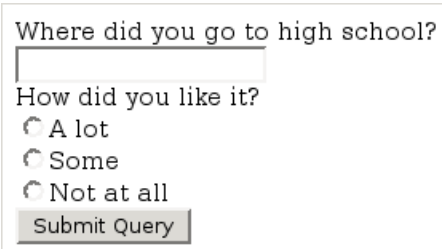

Figure 1. Example minimal HTML from a Topsl survey

Every time a participant fills out a Topsl survey, Topsl creates a response file where that participant's responses will be stored. Response files are a partial mapping from question identifiers (determined statically using the survey's static summary) to responses that represents the answers to all questions that were asked during execution. Similarly, a survey's static summary is a mapping from question identifiers to question text for all questions that could possibly be asked. The static summary of the above survey would look something like:

```
'((q1 "Where did you go to high school?")
  (q2 "How did you like it?")
  (q3 "Where did you go to college?")
  (q4 "How did you like it?"))
```

The question identifiers from the static summary can then be used to create a database table or a statistics file. Using the static summary and any number of response files, Topsl can create comma-separated value files suitable for importing into most typical spreadsheet and statistics programs used by social scientists. With the static summary and a particular response file, Topsl can correlate questions and responses in a browser for easy reading. The static summary is also useful for proofreading question text and for creating printable surveys for off-line participants.

3.2 Page Construction

Since Topsl is a domain-specific language for on-line surveys rather than a web-page construction language, survey authors should be able to express their surveys as surveys and not have to use any HTML or otherwise specify presentation. To support that, Topsl's design factors the presentation of a question from the question itself by making use of values we call page elements (**?** being one example). A page element is a value with a particular meaning in the domain of web surveys (*e.g.* a question, a grouping of questions, or a block of instruction text) that contains information about how to render itself and how to determine what answers a user submitted that correspond to it. The separation is general enough that it allows page elements to represent questions that correspond directly to individual HTML form elements, questions that correspond to multiple form elements, and even question groupings that alter presentation but that do not directly correspond to form elements at all.

In addition to providing several built-in page elements, the language must allow advanced users to create their own: we cannot predict every kind of question any researcher might ever want to ask, so we should not restrict them to only the kinds of questions we thought of when we designed the language.

3.3 Abstract Pages

Surveys frequently ask sets of questions in which each question differs from the other only in very small ways. For instance, the survey in the first example asks the participant the same set of two questions twice, once asking participants whether they enjoyed high school and the next time asking whether they enjoyed college. In the survey we built for Finkel, a block of four questions asking the participant to predict how he or she would feel about a particular topic two weeks, one month, two months, and three months in the future was repeated seven times in the course of the survey with the same phrasing each time, varying only in the topic the questions addressed. In situations like these we need to be able to make an abstraction over a page parameterized over the pieces that vary. Topsl is designed to handle such situations using a form of abstraction that looks syntactically just like a normal Scheme function definition:

```
definition  ::=  ... as before ...
             |   (define (<variable>+) <survey-element>+)
```

We can now use this new **define** syntax to abstract the original example survey.

```
(define enjoy (radio "A lot" "Some" "Not at all"))
(define (where-attended school)
  (page (? free "Where did you go to " school "?")
        (? enjoy "How did you like it?")))
(where-attended "high school")
(where-attended "college")
```

Here *where-attended* is an abstraction over a page parameterized over the school to ask the participant about — it takes a school as input and produces a page as output. The definition of *where-attended* can make use of variables in question text, and the parameterized page can be applied to arguments multiple times to yield multiple different pages. However, despite appearances, **define** in Topsl is not the same as the normal Scheme **define** and does not bind *where-attended* to a normal Scheme procedure. Instead, it defines it as a Topsl procedure that runs at compile-time rather than at run-time, and can only be used in contexts that accept Topsl.

3.4 Dynamic Surveys

In order to write surveys whose question responses affect survey control-flow we need a way to access the question responses while the survey is still running. Topsl allows this with another page element, **?/named**, that allows a programmer to name a question, and **bind**, which binds the response of a named question to an identifier which is accessible by other Topsl code. The **bind** form takes a list of identifiers to bind to named question responses and a page in which the named questions can be found.

We have now seen two ways of displaying pages in example surveys: simple page expressions constructed with the **page** form, and applications, which apply a parameterized page to arguments. To allow both mechanisms of page displaying in **bind** expressions we lift the **page** syntax into a new production, *page-expr*, and make it a new survey element:

```
survey-element  ::=  ... as before ...
                |    <page-expr>
                |    (bind (<variable>*) <page-expr>)
page-expr       ::=  (page <page-element>*)
                |    (<variable> <scheme>*)
```

The **?/named** form is identical to **?** except that it takes an additional identifier as its first argument which will be used as the question's name, significant only to **bind** and to the survey's static summary (which uses it as the question's reported name rather than automatically generating a name for it).

The **bind** form displays the page in its page expression to the participant and then binds the given names to the values the participant supplied in any subsequent expressions. The page is required to provide at least those names extracted by bind. The behavior of questions that exist on the page but which are not answered by the participant is specified by the question type. For this paper, we use only mandatory question types, which Topsl requires the participant to answer or it will redisplay the page. We can use the **?/named** and **bind** forms to construct the following survey:

```
(bind (fav-num)
  (page (?/named fav-num free "What's your favorite number?")))
(page (? free "Why do you like " fav-num "?"))
```

This survey asks a participant two questions, the second of which has text that cannot be determined until the first one is answered. When this survey is visited on the web it will ask the participant for his or her favorite number. When the page is submitted, the next page will have one question which will ask the participant why they like the number he or she submitted, which will be contained in the question text.

Clearly Topsl cannot know the complete text of dynamically-determined questions when generating a static summary. In such cases, the static summary uses the identifier in the question text as a placeholder for the dynamic value. For instance, the static summary of the above survey looks like this:

```
'((fav-num "What is your favorite number?")
  (q2 "Why do you like " fav-num "?"))
```

3.5 Adding Control Flow

If we did not need to support static summaries, then making the **?**, **page** and related Topsl forms behave exactly as normal functions would be ideal. However, were we to make that design decision, static summaries would be impossible to build. For instance, the following program would be legal even though the number of questions on any given page cannot be determined:

```
(define (problem n)
  (if (zero? (random 2))
      '()
      (cons (? yes-no "Is " n " prime?")
            (problem (add1 n)))))
(apply page (problem 0))
```

We could use a static-flow analysis algorithm such as the set-based analysis (SBA) system developed by Meunier *et al* to indicate what values could possibly flow into questions and pages [9], but even the best of those techniques are too conservative for our needs. Using the values obtained from SBA to construct a static summary would generate possibly infinite number of questions that would never actually appear in the survey, making it impossible to construct the static summary in some cases.

To be able to generate static summaries reliably, we restrict the syntax of Topsl so that the contents of every page and every question a Topsl program will display is syntactically apparent (perhaps with information that does not affect the number of questions on a page), and let the static summary include all questions that appear syntactically in the program. This restriction does not eliminate all analysis errors since questions that a Topsl program can never reach will be included in its static summary, we believe errors of that nature will not be important in practice.

The biggest impact of that restriction is that Topsl code cannot intermingle arbitrarily with Scheme code. To ameliorate that situation, Topsl includes its own control-flow forms that enforce syntactic restrictions to allow for analysis but give programmers significant power over the flow of their surveys. We will show two examples of control-flow forms, **when** and **for-each**:

```
survey-element  ::=  ... as before ...
                |    (when <scheme> <survey-element>+)
                |    (for-each <variable> <scheme>+)
```

Both forms behave similarly to their Scheme namesakes. If the test position of the **when** form is a true Scheme value then the consequent Topsl expressions are executed. The **for-each** form takes a variable which must be defined as a parameterized page. All subsequent Scheme expressions must evaluate to list values whose elements will be used as arguments to the parameterized page. For instance, arbitrary Scheme is allowed in the test position of Topsl's **when** form, but the consequents must be Topsl forms. This restriction allows us to easily extract the static summary by listing the questions found in all control paths without having to perform static evaluation of full Scheme to determine what those paths could be.

We can now construct a more interesting survey where the responses affect control flow. The survey in figure 2 uses the multi-select function to create a new question type *countries* which will allow the participant to select any number of countries from an HTML selection box. We then define a page, *about*, which takes a string, *country*, and asks a participant the questions we are interested in. The survey then uses the **bind** form to bind two variables, *england?*, a boolean value from the yes-no question, and *been-to*, which is a list of strings selected from the multi-select box. The survey uses **when** to ask the participant about England if *england?* is true and then loops over the other countries the participant has been to, asking about those.

3.6 Finkel's Loop and the Typical Expression

In the previous example we loop over the return value of a *multi-select* box, but Topsl also needs to be able to loop over values generated from arbitrary computations such as lookup in a database.[2] We support this feature by allowing programmers to define arbitrary Scheme functions and apply them in control-flow forms where

[2]It was this specific feature that professional survey authoring companies could not provide Finkel.

```
(define countries (multi-select "Germany" "France" "Spain"))
(define (about country)
  (page (? yes-no "Did you like " country "?")
        (? yes-no "Is it cold in " country "?")))
(bind (england? been-to)
  (page (?/named england? yes-no "Have you ever been to England?")
        (?/named been-to countries "Where else have you been to?")))
(when england? (about "England"))
(for-each about been-to)
```

```
'((q1 "Have you ever been to England?")
  (q2 "Which other countries have you been to?")
  (q3 "Is it cold in England?")
  (q4 "Did you like England?")
  (q5 "Is it cold in " country "?")
  (q6 "Did you like " country "?"))
```

Figure 2. A complete Topsl survey with static summary

Scheme is allowed. In the following example we use a Scheme function, *get-all-other-countries*, which takes the list of countries the participant was asked about above and returns all the other countries found in a database.

```
;; definition of get-all-other-countries elided
(for-each about (get-all-other-countries been-to))
```

We have now shown both extremes of how a Topsl programmer can use Topsl forms to control flow. In the simple example the programmer branched and looped over the responses of questions. In the most recent example the programmer wrote a complicated Scheme program to loop over. In the authors' experience, the complexity of most conditionals fell somewhere in the middle. It was particularly common to have a nested boolean expression with a few common Scheme predicates on the question results. For example, the code snippet below can be used to decide whether the subject has a passport and then ask pertinent questions. For the sake of example, the authors assume that if a participant has been to more than one country then he or she has a passport.

```
(when (or (> (length been-to) 1)
          (and england? (not (empty? been-to))))
  (page (? free "When did you get your passport?")))
```

Allowing simple expressions like the one in the predicate position above gives Topsl a simple learning curve. It is not necessary for Topsl programmers to understand full Scheme, just the functions they want to use. As a result a Topsl programmer's knowledge can scale with his or her increasingly complex surveys.

3.7 Growing Topsl

Topsl provides the **require** from, taken from mzscheme, which allows Topsl programs to import other modules containing new Topsl forms. The modules the surveys require can be written in Topsl but it is not required. As such programmers can extend the Topsl language in languages other than Topsl itself, including full Scheme. In doing so, the author of that module makes a trade off. He or she gives up the safety ensured by Topsl forms and must take on the responsibility of ensuring that static analysis is maintained and that the new core forms behave as expected. However, he or she is no longer restricted to what Topsl is able to express to create new forms. This mechanism for extending a language using another language, pointed out by Krishnamurthi [8], provides Topsl with unbounded expressibility even the in presence of the static analysis restriction, allowing the language to grow in ways not anticipated by its authors.

4 Implementation

Implementation of the "dynamic" portion of Topsl — that is, the portion that presents web pages to a user and stores that user's responses — is reasonably straightforward. The user's program becomes a servlet for the PLT web server [6]. The server handles session management and other HTTP-related issues and allowed us to think of the web as a normal input/output device, greatly simplifying our development effort. Topsl programs are the composition of Scheme macros and functions that generate XHTML pages encoded as S-expressions and hand them off to the PLT web server for shipping to the users. We prevent programmers from using other Scheme code in arbitrary positions with the PLT module system, which allows us to provide an alternate language for programs.

While this approach works very well, it does have one important problem. The semantics of the PLT web server do not exactly match those we need for Topsl programs: we found that a user answering a question was most naturally modeled with a destructive update to a global record, with the caveat that if a participant hits the back button the answer is erased. Unfortunately, the PLT server does not undo destructive updates to variables when a user hits the back button. However, it does restore the values of lexically-bound variables, so we solved the problem by principled use of state-passing style in our implementation: a record representing the current answers to all questions is passed in to every function that needs to read or alter them. After the survey is completed, the Topsl framework passes this "result monad" to the data-storage module, which writes it out. We were initially worried that this strategy might be too memory-intensive on the server and that an approach in which the framework immediately stored all answers in a database would be necessary, but in practice even our relatively modest server (a Pentium III-800 MHz with 128MB RAM) handled the load with no problems.

Implementation of the static summary feature turns out to be mostly trivial as a result of restricting Topsl's syntax to only Topsl forms. In a survey without abstraction, static analysis is trivial, the summary is essentially just the syntax of that survey minus any Scheme expressions. Allowing parameterized pages means we need to statically expand any parameterized pages where they are applied. To enable this, Topsl provides a special **define** form which behaves differently from the normal Scheme behavior when defining Topsl values. The **define** form expands its body and checks to see if that body expands to a Topsl core form. If it does, the **define** expands to a **define-syntax**. If not, it expands to a regular Scheme **define**. Macros cannot be higher-order; however, parameterized pages are only available from within Topsl, and the Topsl forms such as **for-each** that treat pages as higher-order functions can be written to cope with the altered interface.

5 What Does Scheme Give Us?

Scheme has a powerful macro system that allows us to write Topsl forms in terms of Scheme in a very clear and easy-to-maintain manner while avoiding the need to write a parser and compiler from scratch. Scheme's macro system also provides us two very important additional advantages. First, since Topsl is defined as a collection of macros over Scheme, we get seamless escapes into Scheme without any complications. That is, we do not have to marshal data or define a communication "wrapper" layer for communicating between Topsl and Scheme: under the hood, it's all just Scheme. Second, extending the language with new syntactic forms is a simple process of defining a macro over existing Topsl forms. That allows us to grow our language to meet the unforeseen requirements of future surveys without having to edit the Topsl compiler.

6 What Does PLT Scheme Give Us?

The PLT Scheme suite provides two tools that make our work easier: the PLT module system and the PLT web server.

The PLT module system gives us a flexible way to build languages from other languages [3]. Writing Topsl as a module language gives us the ability to compile Topsl code in any way we choose, taking an entire program at once rather than dealing with one subexpression at a time as we would have to with normal macros. It also allowed us to reuse existing Scheme code in our implementation without having to handle name space collisions.

The continuation-based PLT web server [6] made it much easier for us to make language constructs that query a web user for input. Topsl is an imperative language where presenting a page to a participant is implemented as a call to a function that returns the participant's answers and presents the page to that participant as a side effect. The PLT web server allows us to ignore the complication of web-based communication and implement that feature in a natural way, without worrying about implementing the complicated transformations that would otherwise be necessary to make it work properly [5].

7 Experience

Topsl's first application, and our motivation for developing it, was an on-line survey used in a longitudinal study of dating relationships at Northwestern University. The survey had 70 participants each of whom was asked to visit the survey site once every two weeks and answer some subset of the survey's 248 unique questions that depended on his or her answers from all previous sessions and from the current one: for instance, if the participant had reported that they started dating someone in one session and said they were single in the next, the survey would proceed to a page of questions about the breakup. Also, every time a participant broke up with someone, that person's initials were added to a list; on every subsequent session the survey would present a few questions about each person on that list.

We found Topsl to be an invaluable asset in developing the survey. It let us focus on the survey's particular unique features without needing to worry about our changes introducing bugs in the underlying mechanisms that handled sessions and data storage. For that reason, we were able to develop the survey extremely rapidly given its complexity: we developed prototypes of both the language and the survey in two days, and afterward we were able to modify the survey easily to suit the various revisions its designer requested. For instance, one early revision requested was that we randomize the order in which questions in certain groups were presented to participants. We accommodated that request by writing a Topsl extension that introduced a new page element that randomly shuffled its sub-elements when it presented them on-screen. Data storage and other aspects of presentation were unaffected, so we were able to make the change and be confident of its functionality in only a few hours.

The static summary technique discussed in this paper was developed as a result of failings in that prototype. Our original design required giving every question a unique name and further required a redundant listing of that name in some situations. We found the burden of naming each question quickly became a maintenance nightmare: a request to insert or remove a question would ruin our naming strategy, and changing a name in one place but not another would cause apparent data loss. The survey summary technique avoids this problem while still giving us more than enough flexibility to implement our original survey and every other survey we have seen since.

8 Related Work

There are a considerable number of mechanisms for creating online surveys apart from implementing them in a general-purpose language. Two domain-specific languages, SuML and QPL, stand out as being the closest to the goals the authors set for Topsl.

SuML is an XML/Perl-based survey language in which the programmer describes a survey in an XML document which follows the SuML Schema. The SuML Schema has a **question** element which contains question text and a sequence of allowable responses, much like Topsl. The root **survey** element contains any number of questions and a **routing** element that describes control flow. The **routing** element contains any number of **if** and **ask** elements which are composed to ask questions in the survey and branch on their responses.

The programmer creates two files in addition to the content of the survey: an XSLT stylesheet and a front-end Perl CGI program. The style sheet is responsible for describing what a survey will look like when presented on the web to a participant, and multiple stylesheets can be written for different mediums. The front-end is a Perl CGI program that acts as the entry point to the survey.

SuML's most significant problem for our purposes is its notion of control-flow is very limited, providing its users with only an **if** statement with which to branch to different parts of the survey. Furthermore, the test position of the **if** is written in language for accessing various fields of the XML allowing the programmer to reference question responses. This approach limits control flow to being affected by only responses given in the current survey execution.

In addition, SuML is somewhat too generic for our purposes. The user-written Perl CGI is in charge of driving the survey by using SuML's Perl API to get the next questions to be asked and then present them as well as storing the results of the questions asked. Putting the burden on the programmer makes survey development more difficult, time-consuming, and error-prone.

QPL is another domain-specific language for creating surveys that suffers from very similar problems to SuML's. QPL's semantics are reminiscent of BASIC: it is an imperative language using **if** and **goto** for control flow along with a large set of built-in predicates

used for conditional testing. Current distributions provides users with a large set of comparison functions for use with **if**; however, it lacks an means of growing to meet programmers' changing needs.

9 Further Work

One major avenue of future work we plan on pursuing is making it easier for non-programmers to develop simple surveys in Topsl. While programmers who want a rapid way to develop surveys and are experienced with Scheme should find Topsl intuitive, social scientists who have no programming experience may have difficulty with it. To that end, we suspect that providing a graphical syntax with a WYSIWYG page construction to make the syntax more like word processing would make Topsl more natural for social scientists. Syntax for forms like **when** and **for-each** have been taken from Scheme to meet our programmer audience's expectations of how they should be used, but a graphical syntax that relates pages with flow-control arrows would be more natural for non-programmers. We expect to be able to implement this syntax with the help of PLT Scheme's MrEd toolkit and the substantial graphical editing features of DrScheme.

We would also like to investigate the possibility of adding shared question and page libraries to Topsl. Since social scientists often include the same questions verbatim on multiple surveys to make the surveys more easily comparable, shared libraries are a natural fit. However, they pose some interesting problems: with our current design, for instance, every question whose answer is important to a survey's flow control must be named explicitly in its declaration. In a library, this would not work out well since library authors would have to give every question a name (which is impractical in our experience) or guess which questions will be important to future surveys (which would force users to copy the library and make source code modifications if the library author guesses wrong). A solution to this problem would be quite useful, so we consider it an important topic to investigate.

10 Contributions

We have designed and implemented a survey language system that uses Scheme's capacity to build new languages to solve a pressing problem in many of the social sciences. In the process, we have illustrated the power of building new languages to simultaneously make programs easier to write and less error-prone.

11 Acknowledgments

The authors would like to thank Matthias Felleisen for invaluable guidance throughout the development of this project.

12 References

[1] Barclay, Lober, Huq, Dockery, and Karras. SuML: A survey markup language for generalized survey encoding. In *AMIA Annual Symposium*, 2002.

[2] Michael Birnbaum, editor. *Psychological Experiments on the Internet*. Academic Press, 2000.

[3] Matthew Flatt. Composable and compilable macros. In *ICFP*, October 2002.

[4] R. C. Fraley. *How to conduct behavioral research over the Internet: A beginner's guide to HTML and CGI/Perl*. Guilford, 2004.

[5] Graunke, Findler, Krishnamurthi, and Felleisen. Automatically restructuring programs for the web. In *Automated Software Engineering*, 2001.

[6] Graunke, Krishnamurthi, Van der Hoeven, and Felleisen. Programming the web with high-level programming languages. In *ESOP*, 2001.

[7] Guy L. Steele Jr. Growing a language. *Journal of Higher-Order and Symbolic Computation*, 12:221 – 236, October 1999.

[8] Shriram Krishnamurthi. *Linguistic Reuse*. PhD thesis, Rice University, May 2001.

[9] Meunier, Findler, Steckler, and Wand. Selectors make analysis of `case-lambda` too hard. In *Scheme and Functional Programming*, 2001.

[10] U.S. General Accounting Office. QPL. Software: http://www.gao.gov/qpl/.

[11] Olin Shivers. A universal scripting framework or lambda: the ultimate 'little language'. *Concurrency and Parallelism, Programming, Networking, and Security, Lecture Notes in Computer Science*, 1179:254–265, 1996.