Embedding a Questionnaire DSL with SugarJ

Sebastian Erdweg

TU Darmstadt, Germany

We describe our SugarJ-based solution to the 2013 Language Workbench Competition. As part of this competition, we developed domain-specific language (DSL) for questionnaires that features conditional questions, locally defined questions, derived values, name checking, type checking, checking for overlapping question instances, and basic tool support such as code coloring. Using SugarJ, we have realized the questionnaire DSL as a language extension of Java that translates a questionnaire into a Java Swing component. We have realized the questionnaire DSL via separate components for the syntax, code generation, tool support, name checking, type checking, and overlap checking. Moreover, we use SugarJ's support for layout-sensitive syntax to use indentation instead of parentheses in the design of the questionnaire DSL.

The source code of the questionnaire DSL is available online: `https://github.com/seba--/sugarj/tree/questionnaire/case-studies/questionnaire-language`.

1 Introduction

SugarJ is an extensible programming language [1, 4, 3]. SugarJ is very flexible and allows programmers to introduce arbitrary domain-specific syntax, domain-specific semantics, domain-specific static analyses, and domain-specific editor services. Therefore, SugarJ can accommodate extensions to support a wide range of domains with their specific notation, invariants, and semantics accurately.

Despite this flexibility, SugarJ is also principled. SugarJ organizes language extensions in libraries of the base language, which are activated through regular import statements. By investigating the import statements of a source file, a programmer can modularly reason about the language extensions that are active locally. Furthermore, the organization of language extensions as libraries allows extension developers to decompose an extension into multiple smaller libraries, and even to use language extensions in the definition of other extensions.

SugarJ is realized as a compiler. Given a source file, the SugarJ compiler resolves the dependencies of the source file, activates any imported language extensions, and compiles

the body of the source file to produce a regular Java class file. For the definition of language extensions, SugarJ relies on the declarativity and composability of SDF [9] for the definition of syntactic extensions, Stratego [11] for the definition of semantics and static analyses, and on Spoofax [7] for the definition of editor services. While the SugarJ system supports various base languages, such as Haskell [6], in this work we focus on the Java-based variant.

In the remainder of this paper, we show to use SugarJ for embedding a domain-specific language (DSL) into Java. To this end, we implement a DSL for questionnaires with SugarJ. The questionnaire DSL comprises the following features:

- Domain-specific and layout-sensitive syntax that reuses Java syntax for conditional expressions (Section 2).
- Transformation of a questionnaire into a Java Swing component that visualizes the questions and uses reactive programming to update derived values (Section 3).
- Separately defined analyses for name resolution, type checking, and overlap detection (Section 4).
- Editor support to provide a domain-specific syntax coloring and code folding (Section 5).

2 Questionnaire syntax

We designed a simple layout-sensitive syntax for the questionnaires as depicted in Figure 1. Our questionnaire DSL permits the declaration of questions, derived values, question groups, locally defined questions, and conditionally placed questions. All of these features are locally activated in a source file by importing the library `quest.Language`, which reexports the separately defined syntax, semantics, analyses, and editor services for the questionnaire DSL.

The most central ingredient of a questionnaire are questions. In our DSL, a question declaration has to specify the expected answer type of the question. Our syntax definition allows any Java type for questions, however, our semantics currently only supports the types **Boolean**, **Integer**, and **String**. Following the keyword **question** we require a Java variable identifier that can later in the questionnaire be used to refer to the user's answer. Next we expect the equality operator and finally the question text.

If a question declaration is prefixed by the keyword **define**, the question is locally defined and can be used within the questionnaire by referring to it by name after the keyword **ask**. Value definitions such as `oldEnough` are syntactically similar to questions, but instead of the question text we expect a Java expression after the equality operator. Questions can be organized in groups by declaring a named **question group**. Finally, questions can appear depending on user's answers to previous questions. To conditionally place a question, it can be embedded into an *if-else* statement. As condition for an *if-else* statement, we allow the full class of Java expressions where occurrences of variables refer to previously asked questions.

```
package test;

import quest.Language;

public questionnaire MobileSecurity
  define Boolean question securityRelevant =
    Is security an issue for you?

  Integer question age =
    How old are you?
  Integer value oldEnough = age >= 18
  Boolean question useMobileDevice =
    Do you use any mobile devices?

  if oldEnough && useMobileDevice
    question group deviceDetails
      Integer question howManyDevices =
        How many mobile devices do you use?
      String question whatOS =
        Which operation system are you mainly using?
    ask securityRelevant
  else
    if oldEnough
      Boolean question usePC =
        If you do not use a mobile device,
        are you maybe using a PC?
      if usePC
        ask securityRelevant
```

Figure 1: Example questionnaire in SugarJ,

Syntax definition. SugarJ language extensions are organized in regular libraries of the base language and declared with a *sugar* declaration. For the questionnaire DSL, we decomposed the language definition into separate sugar declarations for syntax, semantics, analyses, and editor services. Figure 2 shows excerpts of the syntax definition for the questionnaire DSL.

SugarJ uses a layout-sensitive extension [5] of SDF [9] for the implementation of syntax. For the questionnaire DSL, the syntax definition is mostly straight forward. We integrate questionnaires as a top-level declaration into the base language Java, which allows us to declare questionnaires in place of, for example, a Java class. A questionnaire consists of a list of question elements QuestElem. In the excerpt of Figure 2, we show the definition of regular questions and conditionals. For example, we specify that a conditional question element ConditionalQuest starts with the keyword **if** followed by a condition in Java-expression syntax. The body of a conditional is again a list of question elements and the else branch is optional.

We designed a layout-sensitive syntax for our questionnaire DSL: We require that

```
package quest.lang;
public sugar Syntax {
  // top-level questionnaire declaration
  context-free syntax
    Quest -> ToplevelDeclaration
    AnnoOrSugarMod* "questionnaire" JavaId QuestList -> Quest {cons("Questionnaire")}

    -> QuestList {cons("QNil")}
    QuestElem QuestList -> QuestList {cons("QCons"), layout(1.first.col == 2.first.col)}
  ...
  // question
  context-free syntax
    Question -> QuestElem {layout(1.first.col < 1.left.col)}
    QuestType "question" QuestId "=" QuestText -> Question {cons("Question")}
    QuestionString -> QuestText {cons("QuestText"), layout(1.first.col <= 1.left.col)}

    JavaID -> QuestId {cons("QuestId")}
    JavaID -> QuestType {cons("QuestType")}

  // conditional question
  context-free syntax
    ConditionalQuest -> QuestElem
    "if" JavaExpr QuestList ConditionalElse? -> ConditionalQuest {cons("CondQuest"),
      layout(1.first.col < 2.first.col && 1.first.col < 3.first.col && 1.first.col == 4.first.col)}

    "else" QuestList -> ConditionalElse {layout(1.first.col < 2.left.col)}
  ...
}
```

Figure 2: Syntax definition for the questionnaire DSL.

all elements on the same nesting level start at the same column. For example, all questions inside a question group must be indented the exact same amount of whitespace. Furthermore, we require that elements filling multiple lines must be indented further from the second line on. For example, the question text must start further to the right than the first line of the question declaration.

To enable the *declarative* specification of layout-sensitive syntax, we extended SDF and its Java-based scannerless generalized LR parser with *layout constraints* [5]. As shown in Figure 2, we denote layout constraints as annotations of regular SDF productions. A layout constraint restricts the applicability of the context-free production it annotates by constraining the relative positioning of subtrees. For example, for question lists QuestList, we require that the head of the list and the tail of the list must be horizontally aligned, that is, they must start at the same column. We use numbers to refer to the subtrees parsed by a production (1 refers to the first subtree). The token selector first selects the first token of a subtree; token selector left selects the left-most token of a subtree. Based

on these tokens, we do arithmetic computation and comparison based on the column and line number of tokens. For example, the constraint 1.first.col < 1.left.col specifies that subtree 1 must adhere to Landin's offside rule [8], that is, all tokens must be further indented than the first token of the subtree.

The syntax for the questionnaire DSL integrates into the Java base language by declaring that a questionnaire can be used where a Java top-level declaration is expected (production Quest -> ToplevelDeclaration). This way, after importing the sugar declaration quest.lang.Syntax or the wrapper quest.Language, questionnaires can be mixed into otherwise regular Java files. Accordingly, the SugarJ parser will produce a syntax tree that contains both Java nodes and nodes of the questionnaire DSL. We define the semantics of the questionnaire DSL by transforming this mixed syntax tree into a syntax tree that only contains Java nodes.

3 Questionnaire semantics

A SugarJ language extension defines the semantics of the extended syntax by translation into the base language (or extensions thereof). For the questionnaire DSL, we generate code that uses the Java Swing API to present the questionnaire to the user. In Figure Figure 3, we show the generated GUI for the MobileSecurity questionnaire from Figure 1.

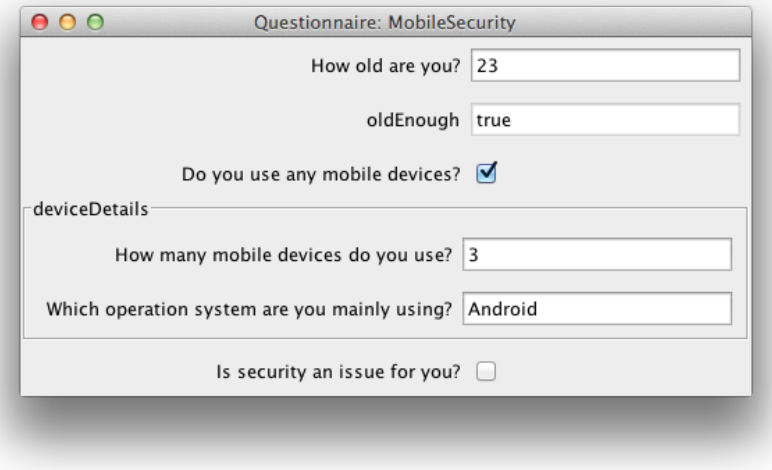

Figure 3: Java Swing GUI generated from the MobileSecurity questionnaire from Figure 1.

We generate the GUI for a questionnaire by separately translating different questionnaire elements into JComponent instances. Depending on the answer type of a question or derived value (**String**, **Boolean**, or **Integer**), we generate JTextField or JCheckBox instances. We group elements of a question group inside a JPanel component with visible border. For conditionals, we group elements of the *then* and *else* branch in two borderless JPanel components whose visibility we dynamically change depending on the condition.

```
package quest.lang;

import concretesyntax.Java;
import quest.lang.Syntax;
import quest.analysis.Typing;

public sugar Transform {
  desugarings
    desugar-questionnaire
    desugar-question
    desugar-derived-value
    desugar-question-group
    desugar-conditional-group
    desugar-local-quest
    desugar-quest-ref
  rules
    desugar-questionnaire :
      Questionnaire(mods, Id(java_x), body) ->
      <put-mods(|mods); put-fields(|fields)>
      java_tdec |[
        class java_x extends javax.swing.JFrame {
          public java_x(String s) { super(s); }
          public static void main(String[] args) {
            javax.swing.SwingUtilities.invokeLater(new Runnable() {
              public void run() {
                java_x frame = new java_x("Questionnaire");
                frame.init();
                frame.setPreferredSize(new java.awt.Dimension(800,600));
                frame.pack();
                frame.setVisible(true);
              }
            });
          }
          public void init() {
            // Add components to the container.
            // Constructs (e.g., conditional quest blocks) may change the current container.
            javax.swing.JComponent container = new javax.swing.JPanel();
            container.setLayout(
              new javax.swing.BoxLayout(container, javax.swing.BoxLayout.Y_AXIS));
            getContentPane().add(container);
            ~block:Block(flat-body)
          }
        }
      ]|
    where <flatten-questlist> body => flat-body
        ; (QuestVars; map(strip-annos); nub) => vars
        ; <map(...)> vars => fields
  ...
}
```

Figure 4: Semantics of the questionnaire DSL.

In Figure 4, we display an excerpt of the questionnaire semantics. SugarJ allows the definition of transformations inside *sugar* declarations. A desugarings block declares transformations that the SugarJ compiler should automatically apply to programs, whenever these transformations have been brought into scope using import statements. For the declaration of transformations themselves, we use the program-transformation language Stratego [11], which like SDF is part of the SugarJ language.

We import three libraries when defining the questionnaire semantics in Figure 4. First, we activate support for program generation using concrete Java syntax [10]. This allows us to generate Java code by writing regular Java code inside brackets |[...]| with escapes ∼. Next we import the questionnaire syntax that declares the constructors of the questionnaire DSL. And third, we import the extension that defines type analysis for questionnaires. We require the typing extension to generate different Swing components depending on a question's answer types.

Finally, we use a simple form of reactive programming to realize the automatic update of derived values and conditions: For each answer, derived value, and condition, we generate an object of the change-observable Variable class. We install change listeners between the variables according to the actual data dependencies, so that changes from a user's answer propagates to relevant derived values and conditions. Furthermore, we let the generated Swing components observe the question, derived value, or conditional they represent, so that the user interface updates accordingly. The separation between backend (inter-dependent variables) and frontend (Swing components observing one variable each) largely simplified the development of the code generator for the questionnaire DSL.

4 Questionnaire static analysis

We define three static analyses for the questionnaire DSL: name resolution, type checking, and overlap detection. While type checking and overlap detection depend on successful name resolution, we were able to define the analyses separately and do not impose a global ordering of their execution.

To this end, we employed a new strategy for the definition of monotonic analyses: Fix-point accumulation of analysis data by chaotic iteration. In this pattern, an analysis does not return a value but installs analysis data (e.g., the value a reference points to or a value's type) as an annotation to the term. We iteratively apply each analysis until none can provide further analysis data, that is, until we reached a fix-point. This pattern elevates the composability of Stratego [2] to enable the implicit composition of inter-dependent analyses, where one analysis can only proceed when another has already computed a required value.

For example, type checking requires name resolution to compute the type of referenced questions. The type checking of a question reference will fail until name resolution has computed and annotated the value that the reference points to. We show the definition of name resolution in Figure 5.

First, we define auxiliary rules to install and retrieve naming-specific annotations. Then, we declare an analysis that scopes the dynamic rules LocalQuest (for locally defined

```
package quest.analysis;
import quest.lang.Syntax;
public sugar Naming {
  rules
    retrieve-reference = get-anno(|"reference")
    put-reference(|t) = rm-anno(|"name-error"); put-anno(|"reference", t)

    retrieve-name-error = get-anno(|"name-error")
    put-name-error(|t) = put-anno(|"name-error", t)

  analyses
    {| LocalQuest, ActiveQuest: analyze-names |}

  rules
    analyze-names = Questionnaire(id, id, analyze-names)
    analyze-names = QNil
    analyze-names = QCons(analyze-names, analyze-names)

    analyze-names = ?Question(_, QuestId(name), _); new-active-name(|name)
    analyze-names =
      ?val; DerivedValue(id, QuestId(?name), analyze-expr-names);
      where(<new-active-name(|name)> val)
    analyze-names = QuestGroup(id, analyze-names)
    analyze-names = ConditionalQuest(analyze-expr-names,
                      analyze-names-local-scope,
                      ?None + Some(analyze-names-local-scope))
    analyze-names = ?LocalQuest(Question(_, QuestId(name), _)); new-local-name(|name)
    analyze-names =
      ?QuestRef(QuestId(name));
      if <LocalQuest> name => ref
        then put-reference(|ref)
        else put-name-error(|"Could not resolve reference.")
      end;
      new-active-name(|name)

    analyze-names-local-scope = {| LocalQuest: analyze-names |}

  rules
    analyze-expr-names = topdown(try(analyze-expr-name))
    analyze-expr-name =
      ?ExprName(Id(name));
      if <ActiveQuest> name => ref
        then put-reference(|ref)
        else put-name-error(|"Could not resolve reference.")
      end

  rules
    new-local-name(|name) = ?t; rules ( LocalQuest : name{t*} -> t)
    new-active-name(|name) = ?t; rules ( ActiveQuest : name{t*} -> t)
```

Figure 5: Name resolution by annotating references.

questions) and ActiveQuest (for asked questions and derived values). The rule analyze-names traverses a questionnaire, registers new names with the dynamic rules, and resolves names of question references. The rule analyze-expr-names resolves expression variables inside the definition of derived values and the condition of a conditional question.

Similar to name resolution, the definition of type checking and overlap detection traverses the questionnaire and installs analysis data. However, these analyses depend on the outcome of name resolution. For example, to type-check a question reference, we retrieve the referred element, extract its type, and install this type as the type of the reference:

```
analyze-types =
  ?QuestRef(_);
  where(retrieve-reference; retrieve-type => type);
  put-type(|type)
```

Due to the dependency on name resolution, type checking of references will be delayed until name resolution has installed a pointer to the referred element. Chaotic fix-point iteration then ensures that we in fact derive all analysis data. In future work, we plan to replace chaotic iteration with a scheduler that dynamically orders static computation to minimize overhead.

```
package quest.lang;
import quest.lang.Syntax;
public editor services Editor {
  folding
    Quest
    Question
    QuestGroup
    ConditionalQuest

  colorer
    darkgrey = 100 100 100
    keyword = 127 0 85 bold

    QuestText : blue
    QuestId : darkgrey
    QuestType : keyword
}
```

Figure 6: Definition of simple editor services for the quesitonnaire DSL.

5 Questionnaire IDE support

SugarJ uses Spoofax [7] to provide domain-specific editor services on a file-by-file basis [3]. To this end, SugarJ developers can define extended editor services in regular libraries,

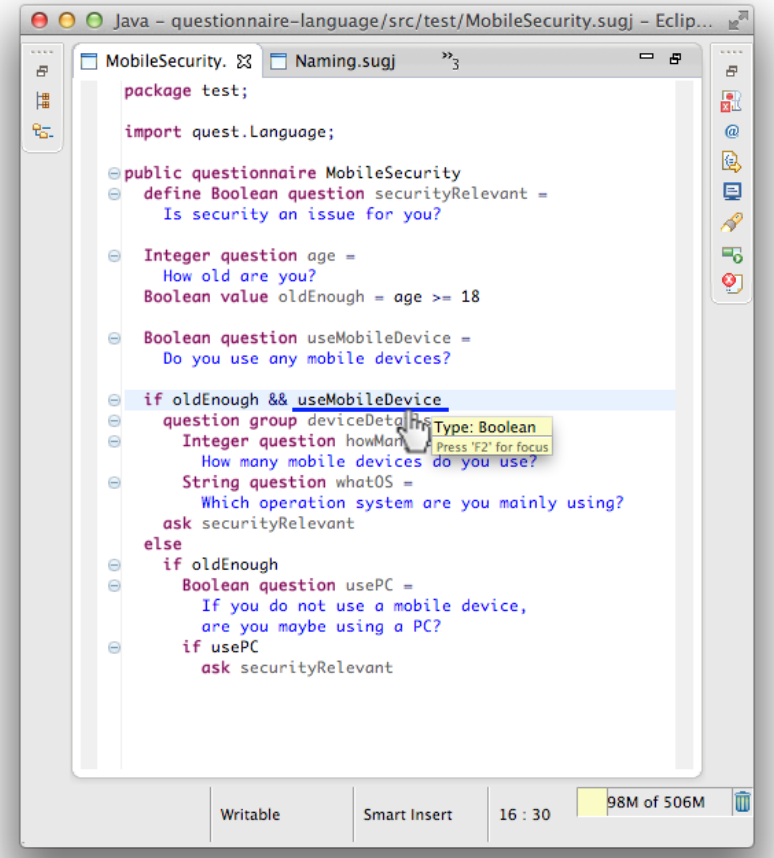

Figure 7: Editor support for the development of questionnaires: code coloring, code folding, reference resolution, type information.

and activate these services by importing the library locally. For the questionnaire DSL we define simple editor services in the library quest.lang.Editor shown in Figure 6.

In addition, based on the static analyses, we define reference resolution (CTRL-click) and show type information in hover-help pop-ups. In the definition of these editor services, we simply extract the relevant analysis data from the syntax tree using retrieve-reference etc. We illustrate the resulting editor for the questionnaire DSL in Figure 7.

6 Conclusion

We have shown how to embed a DSL for questionnaires in SugarJ. We have defined domain-specific syntax, semantics, analyses, and editor services. In particular, we were able to

modularly define analyses by accumulating analysis data in a fix-point computation.

References

[1] S. Erdweg. *Extensible Languages for Flexible and Principled Domain Abstraction*. PhD thesis, Philipps-Universiät Marburg, 2012.

[2] S. Erdweg, P. G. Giarrusso, and T. Rendel. Language composition untangled. In *Proceedings of Workshop on Language Descriptions, Tools and Applications (LDTA)*, pages 7:1–7:8. ACM, 2012.

[3] S. Erdweg, L. C. L. Kats, T. Rendel, C. Kästner, K. Ostermann, and E. Visser. Growing a language environment with editor libraries. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, pages 167–176. ACM, 2011.

[4] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: Library-based syntactic language extensibility. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 391–406. ACM, 2011.

[5] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Layout-sensitive generalized parsing. In *Proceedings of Conference on Software Language Engineering (SLE)*, volume 7745 of *LNCS*, pages 244–263. Springer, 2012.

[6] S. Erdweg, F. Rieger, T. Rendel, and K. Ostermann. Layout-sensitive language extensibility with SugarHaskell. In *Proceedings of Haskell Symposium*, pages 149–160. ACM, 2012.

[7] L. C. L. Kats and E. Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 444–463. ACM, 2010.

[8] P. J. Landin. The next 700 programming languages. *Communication of the ACM*, 9(3):157–166, 1966.

[9] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

[10] E. Visser. Meta-programming with concrete object syntax. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, volume 2487 of *LNCS*, pages 299–315. Springer, 2002.

[11] E. Visser, Z.-E.-A. Benaissa, and A. P. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 13–26. ACM, 1998.