1 Introduction

Provide a brief summary and some background of the domain. Why is a DSL needed, and how would it be useful? What notations/languages, if any, are currently used? What are their strengths and limitations?

2 Users

How is the DSL going to be used, and what specifically is it used for? What kind of users are affected by the DSL and in which way? Who is writing programs and who is using them? What is known or can be reasonably assumed about the (technical/domain) background of users? (Usually, it is safe to assume that users are domain experts.)

Specifically, what are the limitations of the DSL? What cannot be expressed? If different kinds of users are involved, what are there limitations of for the different user groups?

3 Outcomes

What is (are) the outcome(s) or resulting values of executing a DSL program? Also, what kind of (static) analyses of DSL programs are conceivable? Typical examples are type checking or some other form of consistency checking. Note that, in general, a DSL program can produce a variety of results. In fact, this aspect makes the use of a DSL very attractive, and this questions is an important part of the analysis of the DSL’s purpose.

Consider, for example, a DSL for describing vacation plans (including alternatives). We can imagine that a vacation plan can be analyzed to yield a set of date/time ranges for the vacation (based on the availability of flights, hotels, etc.), or a set of cost estimates (based on travel and date options). Moreover, we can imagine deriving to-do lists from the travel plan (making reservations, buying tickets, etc.) or entertainment suggestions (parks, museums, events, etc.).

The outcomes should be made more precise in Section 7 in the form of function signatures once the basic objects and combinators have been identified in Sections 5 and 6, respectively.
4 Use Cases / Scenarios

Describe several typical example problems or use cases that can be expressed in the DSL. For each example, give a summary of what the example is about, and explain how important and representative it is for the domain.

Then describe in more detail the steps involved in solving the problem (either by hand or with existing tools). This process can be very helpful in shedding light on what kind of basic objects, types, and combinators the DSL will have to offer.

The final version of this section should contain a trivial example that illustrates the basic structure of your DSL as well as a bigger example that demonstrates what your DSL can do. The bigger example allows a reader to judge the scope and impact of your DSL and whether your DSL would be useful to them.

It is very important to clearly describe and distinguish between the following aspects.

1. The DSL program for the example
2. How the program is executed
3. What are the inputs to running the example program
4. What are the outputs

5 Basic Objects

What are the basic objects that are manipulated and used by the DSL? Basic objects are those that are not composed out of other objects.

As a general rule, the fewer basic objects one needs, the better, because the resulting DSL design will be more concise and elegant. The basic objects should be described by a set of Haskell type and data definitions. Use the program environment to show code, as illustrated below.

\[
\text{type Point} = \text{(Int,Int)}
\]

Show how (some (parts) of) the examples from Section 4 will be represented by values of the envisioned types. For example:

\[
\text{home :: Point}
\]
\[
\text{home = (10,13)}
\]

Also, list current limitations that you expect in a future iterations to overcome.

Be prepared to revise this section many times!

6 Operators and Combinators

Identify operators that either transform objects into one another or build more complex objects out of simpler ones. Depending on what implementation or form of embedding will be chosen, operators may be given as constructors of data types or functions.

Combinators are higher-order functions that encode control structures of the DSL. The function map is a combinator that realizes a looping construct for lists. The operations of the parser library Parsec are called parser combinators since parsers themselves are represented as functions. The identification of the right set
of combinators is a key step in the design of the DSL.

With basic objects, operators, and combinators, you should be able to demonstrate how the examples from Section 4 can be represented. All limitations encountered here should be classified as either:

(1) Temporary
(2) Fundamental

Temporary limitations should be noted in this section and in Section 5 as TO DO items for future revisions of the design. Fundamental limitations should be reported and listed in detail in Section 2.

Be prepared to revise this section many times!

7 Interpretation and Analyses

This section is optional. Include it if your DSL can produce other outcomes besides the main outcome.

For example, if your DSL offers a type checker or some other static analyses for DSL programs, this section is the place to describe those. Provide a precise description of the different outcomes of the DSL. This information is typically provided by Haskell function signatures (that is, function names and their types).

8 Cognitive Dimension Evaluation

This section is optional. Include it only for non-trivial observations about the notation of your DSL.

An assessment of cognitive dimension of your notation, such as closeness of mapping, viscosity, hidden dependencies, and others, will help you with the re-design of the DSL.

Note that it is advisable to think about the cognitive dimensions constantly during the design of your DSL. Cognitive dimensions are, in fact, a language design tool, and not so much intended to evaluate languages after the fact.

9 Implementation Strategy

Discuss how the advantages and disadvantages of a deep or shallow embedding play out in your DSL.

You could also discuss (but don’t have to): What advantages would an implementation in a language workbench have? Would it be worth the effort?

10 Related DSLs

Try to find DSLs that are similar to the one described here and compare your DSL with those. Note that similarity can be understood as topical as well as technical similarity.

Topically similar DSLs are DSLs for the same or a closely related domain. They have in principle the same or slightly different outcomes, but they may be implemented differently. These DSLs help you refine the design of the DSL requirements described in Sections 2 and 3.

Technically similar DSLs are Haskell DSLs whose types and functions are similar to the ones used in
Sections 5 and 6. They can help sharpen the DSL modeling and implementation described in those sections. Ideally, you can find both, topically and technically similar DSLs. Be sure to properly cite the DSLs as references.

These DSLs can be part of your in-class presentation.

11 Design Evolution

Include this section only if you have something interesting to say.

As you iterate over different designs of your DSL, it is quite instructive to document some of the old, obsolete designs, that is, show the type definitions and function signatures, explain why this design seemed attractive at first and then what motivated you to change it.

This part may seem like an unnecessary burden to you, but it helps you and others to understand your current design, and it probably answers questions that users (or reviewers) of the DSL might have about the design, because they may have thought of your initial design also and are wondering why it has not been adopted.

12 Future Work

A speculation about what it takes to remove some of the limitations and whether it seems worth the effort.

Moreover, what would be the concrete benefits to extend a shallow DSL into a deep DSL? Or, would it be helpful to create an external DSL? What role could a visual syntax or a GUI interface play?

References

List references to similar DSLs identified in Section 10 and potentially other related work.