Chapter 13
Semantics-Driven DSL Design

Martin Erwig
Oregon State University, USA

Eric Walkingshaw
Oregon State University, USA

ABSTRACT

Convention dictates that the design of a language begins with its syntax. The authors of this chapter argue that early emphasis should be placed instead on the identification of general, compositional semantic domains, and that grounding the design process in semantics leads to languages with more consistent and more extensible syntax. They demonstrate this semantics-driven design process through the design and implementation of a DSL for defining and manipulating calendars, using Haskell as a metalanguage to support this discussion. The authors emphasize the importance of compositionality in semantics-driven language design, and describe a set of language operators that support an incremental and modular design process.

INTRODUCTION

Despite the lengthy history and recent popularity of domain-specific languages, the task of actually *designing* DSLs remains a difficult and under-explored problem. This is evidenced by the admission of DSL guru Martin Fowler, in his recent book on DSLs, that he has no clear idea of how to design a good language (2010, p. 45). Instead, recent work has focused mainly on the *implementation* of DSLs and supporting tools, for example, through language workbenches (Pfeiffer & Pichler, 2008). This focus is understandable–implementing a language is a structured and well-defined problem with clear quality criteria, while language design is considered more of an art than an engineering task. Furthermore, since DSLs have limited scope and are often targeted at domain experts rather than professional programmers, general-purpose language design criteria may not always be applicable to the design of DSLs, complicating the task even further (Mernik et al., 2005).

Traditionally, the definition of a language proceeds from syntax to semantics. That is, first a syntax is defined, then a semantic model is decided upon, and finally the syntax is related to the semantic model. This widespread view is reflected in the rather categorical statement by Felleisen et al. that the specification of a program-

DOI: 10.4018/978-1-4666-6042-7.ch013

ming language starts with its syntax (2009, p. 1). This view has been similarly echoed by Fowler, who lists defining the abstract syntax as the first step of developing a language (2005) (although he puts more emphasis on the role of a "semantic model" in his recent book (2010)).

In this chapter we argue for an inversion of this process for denotationally defined DSLs, where the semantic domain of the language is identified first, then syntax is added incrementally and mapped onto this domain. We argue that this *semantics-driven* approach to DSL design leads to more principled, consistent, and extensible languages. Initial ideas for semantics-driven DSL design were developed in our previous work (2011). This chapter expands these ideas and explains the process and the individual steps in detail.

Syntax-Driven Design

We begin by demonstrating the traditional syntax-driven approach, both for reference and to demonstrate how it can lead to a rigid and idiosyncratic language definition. Consider the design of a simple calendar DSL for creating and managing appointments. We first enumerate some operations that the DSL should support, such as adding, moving, and deleting appointments. It should also support basic queries like checking to see whether an appointment is scheduled at a particular time, or determining what time an appointment is scheduled for. One advantage of the syntax-driven approach is that it is easy to get off the ground; we simply invent syntax to represent each of the constructs we have identified. A syntax for the basic calendar operations is given in Box 1, where *Appt* represents appointment information (given by strings, say) and *Time* represents time values.

```
Op  ::=  add Appt at Time
       |   move entry at Time to Time
       |   delete Time entry
```

The add … at … operation adds an appointment at the specified time, move entry at … to … reschedules the appointment at the first time to the second, and delete... entry removes the appointment at the given time from the calendar. A program defining a calendar consists of a sequence of such operations.

```
Prog ::=  Op*
```

With an initial syntax for our calendar DSL in place, we turn our attention to defining its (denotational) semantics. This process consists of finding a semantic domain that we can map our syntax onto, then defining a valuation function that represents this mapping. Looking at our syntax, we can observe that an array-based representation of calendars will yield constant-time implementations of each of our basic operations. Therefore we choose *dynamic arrays* (Schmidt, 1986, Ch. 3) as a semantic domain. A dynamic array is a function that maps elements of a discrete domain to some element type that contains an error (or undefined) element, say ε. In our example, we use the type *Cal* as an instance of dynamic arrays in which the discrete domain is *Time*, and the element is appointment information *Appt*. The semantic domain of dynamic arrays is a semantic algebra that offers operations for accessing and updating arrays. Accessing an element at position t in an array c means to apply the function that represents the array and is thus simply written as $c(t)$. The semantic *update* operation is defined in Box 2.

The semantics of an operation *Op* is a function from one calendar array to another and is captured by a valuation function $[[\cdot]] : Op \to (Cal \to Cal)$, which is defined using the operations from the semantic algebra (Box 3).

The semantics of a calendar program is then defined as the accumulation of the effects of the individual calendar operations, applied to the initial array that is undefined everywhere, that is, $[[\cdot]] : Prog \to Cal$: