Semantics-Driven DSL Design*

Martin Erwig and Eric Walkingshaw
School of EECS, Oregon State University, USA

ABSTRACT

Convention dictates that the design of a language begins with its syntax. We 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. We 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. We 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 programming 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.

* This work is partially supported by the Air Force Office of Scientific Research under the grant FA9550-09-1-0229 and by the National Science Foundation under the grant CCF-0917092.

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 below, 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 as follows.

$update : Time \times Appt \times Cal \to Cal$
$update(t, a, c) = c - \{(t, c(t))\} \cup \{(t, a)\}$

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.

$[[\texttt{add}\ a\ \texttt{at}\ t]]\ c = update(t, a, c)$
$[[\texttt{move entry at}\ t\ \texttt{to}\ t']]\ c = update(t', c(t), update(t, \varepsilon, c))$
$[[\texttt{delete}\ t\ \texttt{entry}]]\ c = update(t, \varepsilon, c)$

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, $[[\cdot]] : Prog \to Cal$.

$[[o_1, o_2, \ldots, o_n]] = [[o_n]]\ (\ldots\ [[o_2]]\ ([[o_1]]\ \{(t, \varepsilon) \mid t \in Time\})\ \ldots)$

The array representation works very well as a semantic domain for the syntax we have defined. It also supports the queries we identified at the start: checking whether an appointment is scheduled at a particular time is just an index into the calendar array, and looking up the scheduled time of an appointment is a linear search. Its weaknesses only become apparent as we extend the language with new syntax and functionality.

First, we consider an extension of the language with a notion of appointment lengths, which will allow us to more accurately determine whether a particular time is available. To do this, we add an extra value to the add construct, an integer representing the number of time slots the appointment will last.

Op ::= `add` *Appt* `at` *Time* `with length` *Int*
| …

The semantics of the modified `add` operation can then be defined by assigning the appointment to each time slot in the calendar that the appointment occupies.

[[`add` a `at` t `with length` n]] $c = update(t + n - 1, a, \ldots update(t + 1, a, update(t, a, c)) \ldots)$

Note that although we have not changed the structure of our semantic domain, its use is now sufficiently changed that we have to redefine the valuation function of both other operations in our DSL as well since one appointment now will generally occupy several array slots. In other words, this extension is not modular, requiring a complete redefinition of our DSL's semantics.

As a second example, consider the extension of the DSL to support overlapping appointments. Note that this extension can be supported perfectly well by our existing syntax, but not by the flat appointment-array representation we have chosen as its semantic domain. We might update our semantic domain to support this feature by considering a calendar to be a time-indexed array of *lists* of appointments. Instead of directly assigning appointments to the array, our valuation function now adds and removes appointments from the list at each time slot. Again, this extension forces us to reconsider our semantics and completely redefine our valuation function.

That both of these seemingly minor extensions cannot be implemented in a modular way suggests that our initial calendar DSL was not very extensible. The problem lies mainly in our choice of semantic domain, which was chosen because it supported a nice and efficient implementation of our initial syntax. Ultimately, the semantic domain we choose has a profound impact on the quality of the language going forward—it gives terms in the language meaning, and so is the foundation on which the syntax is built. In the syntax-driven approach, this important decision is relegated to supporting an after-the-fact definition of some possibly idiosyncratic initial syntax.

Semantics-driven design

The semantics-driven approach begins instead with the identification of a small, compositional semantics core, then systematically equips it with syntax. We argue that considering the semantic domain of a language first leads to a more principled language design, in part, because it forces language designers to begin by carefully considering the *essence* of what their language represents. With the proper semantics basis, the language can be systematically extended with new syntax as new features are added.

It is of course still possible to identify a poor semantic domain when beginning with semantics, just as beginning with syntax does not doom one to a poor language design. The semantics-driven approach we describe in this chapter is not mechanical and still requires creativity and insight on the part of the DSL designer. It does, however, provide much-needed structure to the language design process, and emphasizes the importance of compositionality, reuse, and extensibility in DSL design.

Semantics-driven design is fundamentally incremental and domain-focused, while syntax-driven design is more monolithic and feature-focused. In the syntax-driven approach, designers begin by

anticipating use cases and inventing syntax to implement corresponding features. This is problematic since it is difficult to foresee all cases, leading to an incomplete syntax and idiosyncratic semantics that must be extended in an ad hoc way. In the semantics-driven approach, designers begin by trying to identify a more general representation of the domain, then extend this in a structured way to support specific features.

Semantics-driven language design is also a mostly compositional process that leads naturally to compositional languages. That is, bigger languages can be defined by systematically composing or extending existing smaller languages. This supports the incremental development of DSLs and promotes the reuse of small DSLs in larger ones. Moreover, it supports the decomposition of a DSL's domain into simpler subdomains, making it easier to reason about and identify good semantics domains. Compositionality also makes semantics-driven design less ad hoc than the traditional approach. Compositional development produces a clear account of the individual components of the language and how they are related, making the design easier to reuse, extend, and maintain. This process also produces, as a byproduct, a library of smaller DSLs that can be reused in future language designs. Compositionality is especially important in the context of DSLs since the final language must often be integrated with other languages and tools.

Rest of this chapter

Our research background is programming languages and functional programming, and we therefore approach and discuss the problem of DSL design from a different perspective than the majority of the chapters in this book. In the next section we take some time to describe the programming language view of DSLs and language design, and establish the terminology that will be used throughout the chapter. This will hopefully make the chapter accessible to as wide an audience as possible. We also introduce in this section the strongly-typed functional language Haskell (Peyton Jones, 2003) as a metalanguage for DSL design (Thompson, 2011).

In the third section we use these tools to describe semantics-driven DSL design in detail. We first present a high-level overview of the design process, then proceed by example, demonstrating the (re)design of the calendar DSL using the semantics-driven approach. We do this by first describing the process of decomposing and defining the semantic domain, which is followed by incrementally adding syntax.

The compositional nature of semantics-driven design leads to a view of the language design space in which languages are composed of smaller, mini-DSLs. The relationships of these mini-DSLs and the ways that they can be composed can be captured in *language operators*, which are discussed in the fourth section. The use of language operators supports a structured approach to language design and promotes language reuse.

In Related Work, we relate semantics-driven design to other language design strategies and also discuss other related work not discussed in the main body of the chapter. Conclusions will be presented in the final section.

BACKGROUND

Research on DSLs has been pursued mainly by two different communities: the modeling community and the programming languages (PL) community. Each brings a different background to the area and has developed its own specific views on DSLs. This often involves specialized terminology, a distinctive focus on particular goals, and consequently the use of different methods. Together these two approaches may provide deeper insights through a diversity of perspectives, but it also balkanizes the research area, making it harder for the different groups to talk to one another and potentially limiting progress in the field as a whole.

The purpose of this section is to explain the basic elements of the PL approach to DSLs and to acquaint the reader with the corresponding terminology and methods. This is necessary since the model-based view seems more dominant today—it is the view taken by Fowler (2010) and also the one found in most of the chapters in this book. This section will therefore explain the idiosyncrasies of the PL approach, in order to make this chapter accessible to a broader audience.

Language Structure of DSLs and the Role of Metalanguages

The two major aspects of any language are its *syntax* and *semantics*. (Language pragmatics is also an important aspect, but is probably less relevant in the design phase of languages.) In the PL approach, the syntax of a DSL is usually described by a context-free grammar, rather than by a metamodel. The semantics can be defined in several different ways. Two widely used methods are operational (Pierce, 2002) and denotational (Mitchell, 1998) semantics. A denotational semantics (which we focus on here) consists of two parts: (1) the *semantic domain*, which is a collection of semantic values and operations, and (2) the *valuation function* (or just *valuation*), which is a mapping from the syntax to the semantic domain (Schmidt, 1986). A semantic domain is, in principle, very similar to the notion of a *semantic model* as described in (Fowler, 2010).

Except for purely theoretical treatments of languages, the syntax and semantics of a DSL are commonly expressed in terms of a programming language that effectively acts as a metalanguage with respect to the defined DSL. In this case, the DSL syntax and semantics are defined using constructs of the metalanguage. Exactly how this is done and which constructs are used depends not only on the chosen metalanguage, but also on the implementation style of the DSL. An *external* DSL is a standalone language, which is parsed and interpreted by the metalanguage. In contrast, an *internal* DSL exists within the metalanguage itself, using metalanguage constructs as DSL syntax directly. Internal DSLs are also called domain-specific *embedded* languages (DSELs) (Hudak, 1998) and can be further classified into two embedding styles, *deep* or *shallow*. This will be described in detail later and is also explained in Chapter 19.3 of Simon Thompson's book (2011).

The purpose of this section is not to give a comprehensive overview and comparison of all possible variations on this theme, but only to provide the necessary background and context for the particular language-based approach employed in this chapter. We will therefore consider in the following only internal DSLs (DSELs) and their denotational semantics. Moreover, we will describe the perspective from the point of view of a typed, functional metalanguage.

For concreteness, we use Haskell (Peyton Jones, 2003) as our metalanguage; it has a long tradition as a metalanguage and has been used extensively and successfully to define a wide range of DSLs. We discuss some examples of these in the Related Work section, and many other examples are listed in the Additional Reading section at the end of this chapter. However, much of the discussion applies also to other languages.

Types as Semantic Domains

The values of the semantic domain are naturally given by values of the metalanguage. These values are elements of predefined or user-defined types or data types, and these types therefore define the semantic domain of the DSEL. The meaning of a type such as `Int` is obvious, but the meaning of a data type might not be so widely known. A data type consists of a set of constructors that each have a name and zero or more argument types. For example, the following definition introduces a data type with three constructors for representing pictures containing lines and circles.

```
type Point = (Int,Int)

data Pic = Line Point Point
         | Circle Point Int
         | Pic :+: Pic
```

The `type` definition simply introduces a new name `Point` for the type of integer pairs. This type is used as an argument type in two of the constructor definitions. Each of the shown constructors takes values as indicated by the argument types and builds a value of type `Pic`. For example, the first constructor `Line` builds a picture consisting of a single straight-line segment given by two endpoints. The second constructor `Circle` builds a picture consisting of a circle with the given point value as center and the integer value as its radius. Finally, the symbolic infix constructor `:+:` builds a `Pic` value by overlaying two other pictures.

A semantic domain, such as `Pic`, together with its operations (`Line`, `Circle`, `:+:`, and the pairing operation of points) forms a *semantic algebra* (Schmidt, 1986). Below is an example semantic value (a value of the `Pic` data type) that represents a picture consisting of two concentric circles and an intersecting line.

```
ctr :: Point
ctr = (3,2)

pic :: Pic
pic = Line (1,0) (5,3) :+: Circle ctr 4 :+: Circle ctr 5
```

The definition consists of two smaller definitions. First, a point value is bound to a variable `ctr`. Second, a picture value is bound to `pic`, using `ctr` as the center for the two circles in the picture. The first line of each variable definition is optional and indicates the type of the defined variable (if omitted, it will be inferred by the type checker). The second line provides the definition of the variable.

Built-in and user-defined types and data types are employed as semantic domains in the definition of DSELs. The semantics of the DSEL is then given by a valuation from the syntax to the semantics domain. What this syntax looks like, and how this mapping is realized, depends on the embedding style of the DSL.

An Embedding-Dependent Notion of Syntax

As mentioned above, internal DSLs can be either deeply or shallowly embedded (Thompson, 2011, Ch. 19.3). While semantic domains and semantic values are represented the same in either style of embedding, the representation of DSL syntax is quite different.

Deep embedding

In a deep embedding, the (abstract) syntax of a DSEL is represented explicitly by a data type. Each constructor represents a grammar production (that is, an operation) of the language, and its argument types correspond to nonterminals that occur on the right-hand side of the production. Constructors without arguments represent terminal symbols, and constructors with basic type arguments (such as `Int`) form the link to the lexical syntax. The semantic domain of the represented language is captured by a separate (data) type, and some function `sem` acts as the valuation function, mapping syntactic values to semantic values.

Consider, for example, a DSL for describing pictures that contains commands for drawing lines and right triangles. The syntax for this DSL can be described by a context-free grammar. In the following,

Cmd is the non-terminal ranging over drawing commands, while *Point* and *Num* range over points and integers, respectively.

```
Cmd  ::= line from Point to Point
       | triangle at Point width Num height Num
       | Cmd; Cmd
       | …
```

This grammar excerpt can be directly translated into the following data type.

```
data Cmd = Line' Point Point
         | Tri Point Int Int
         | Seq Cmd Cmd
```

The constructor `Line'` represents a line between the given points,[1] the constructor `Triangle` represents the triangle, and the constructor `Seq` represents the sequential composition of commands. Note how the argument types of the constructors mirror the nonterminals in the corresponding grammar productions. A constructor name together with its the argument types distills the essential components of a grammar production and omits keywords such as `from` or `height`. In this way, data types represent the *abstract* syntax of languages, rather than the concrete syntax.

To define the semantics of the picture language, we employ the data type `Pic` as the semantic domain, then define a valuation function `sem`. We define `sem` by equation, using pattern matching. Each equation maps a case of the syntax (a constructor of `Cmd`) onto a corresponding semantic value (of type `Pic`).

```
sem :: Cmd -> Pic
sem (Line' p1 p2)     = Line p1 p2
sem (Tri p@(x,y) w h) = Line p (x,y+h) :+: Line p (x+w,h) :+: Line (x,y+h) (x+w,y)
sem (Seq d d')        = sem d :+: sem d'
```

The first equation is trivial, directly mapping a syntactic line onto a semantic one. In the second equation we have used a so-called "as-pattern" `p@(x,y)` that matches the complete point value to `p` and at the same time matches the components of the pair to the variables `x` and `y`. The valuation maps a triangle to three lines that are combined into one `Pic` value using the overlay constructor. Finally, a sequence of drawing commands is mapped to an overlay of the corresponding `Pic` values obtained for the two commands.

Shallow embedding

In a shallow embedding, we do not define a data type for the syntax at all, but rather take the constructors of the semantic domain immediately as operations of the DSL. For example, in our picture-drawing DSEL we have as part of the semantic domain the constructor `Line`, which represents a semantic value and can thus be used directly as an operation of the DSL. If we are not satisfied with this syntax, we can always introduce a function definition to change the syntax.

Suppose we want the syntax to be closer to the concrete syntax given in the grammar, enforcing the use of keywords. We can define a function `line` that takes additional keyword arguments, as follows. First we define the keywords that we need as strings of the same name.

```
type KW = String
from = "from"
to = "to"
```

[1] We cannot use the constructor name `Line` since it has been used already in the `Pic` data type.

Language Aspect		Representation in Metalanguage			
		Deep Embedding		*Shallow Embedding*	
syntax	L	data type	`L`	function LHS	`f pat :: T`
program	$p \in L$	value	`p :: L`	expression	`p :: D`
semantic domain	D	(data) type	`D`	(data) type	`D`
semantic value	$v \in D$	value	`v :: D`	value	`v :: D`
valuation	$[[\cdot]] : L \to D$	function	`sem :: L -> D`	function RHS	`rhs :: D`
syntax + semantics	$(L, [[\cdot]])$	data type + function	`(L,sem)`	function	`f :: T -> D` `f pat = rhs`

Figure 1: *Summary of the language-based view of DSLs and the representation of internal DSLs within a typed, functional metalanguage.*

Then we extend the function definition for `line` to take additional `KW` arguments and check, using pattern matching, that the correct keywords have been used in a call of `line`.

```
line :: KW -> Point -> KW -> Point -> Pic
line "from" p "to" q = Line p q
line "to" p "from" q = Line q p
line _ _ _ _ = error "Incorrect keyword!"
```

As illustrated in the function definition, it is very easy in this approach to extend the syntax on the fly, for example, with alternative orderings of arguments. (It is also easy to extend this definition to produce more elaborate error messages that report the incorrect keywords.) If we write a command for drawing a line we have to use the keywords `from` and `to`. If we do, a semantic `Line` value is produced correctly, if we don't, the function reports a syntax error.

```
> line from (1,1) to ctr
Line (1,1) (3,2)

> line to (1,1) to ctr
*** Exception: Incorrect keyword!
```

In addition to the constructor names of the semantic domain (and potentially added syntactic sugar), we also introduce function definitions for those operations of the DSL that are not directly represented by constructors of the semantic domain. The operation for drawing triangles is such an example. The corresponding function definition looks as follows.

```
triangle :: Point -> Int -> Int -> Pic
triangle p@(x,y) w h = Line p (x,y+h) :+: Line p (x+w,h) :+: Line (x,y+h) (x+w,y)
```

As with the command for drawing lines, we could extend the above function definition by arguments representing keywords to enrich the concrete syntax.

The important observation here is that these function definitions are comprised of two parts that combine the definition of DSEL syntax *and* semantics. First, the function head, that is, the left-hand sides of the equations, with the name of the function and its argument patterns, defines the DSEL syntax. Second, the expressions on the right-hand sides of the equations define the semantics of that particular syntactic construct. Since we obtain different function definitions for different operations of the DSEL, the valuation function from syntactic elements to values in the semantic domain is spread across several function definitions.

A summary of the preceding discussion is presented in Figure 1.

Which Embedding for Semantics-Driven Design?

The semantics-driven design process can be used with either implementation strategy, although there are many trade-offs involved. The biggest trade-off between the two embedding styles is in the dimension of extensibility. Deep embeddings directly support the addition of new semantic interpretations of the language. For example, we might want to perform some static analyses on our language or generate a visualization of the program. A new semantic interpretation can be added by simply identifying the type of the new semantic domain, `D'`, and writing a new valuation function that maps values of `L` onto values of `D'`. However, extending a deep embedding with new syntactic constructs is relatively difficult—not only must we extend the `L` data type with new constructors, but we must extend the definition of every function that manipulates or interprets `L` as well.

Conversely, adding new syntax to a shallow embedding is very easy—we just add a new function that generates a value of the semantic domain `D`. Adding new semantic interpretations, on the other hand, is much more difficult in a shallow embedding (and is often incorrectly described as impossible). In order to add a new interpretation onto a type `D'`, we must first extend the semantics domain of the language from `D` to the product of `D` and `D'`, then extend every syntactic function to produce values of this new type. We can then obtain the desired semantic interpretation by simple projections.[2]

We will use the shallow embedding strategy in this chapter since it supports a more incremental style of language development and more closely matches the compositional process described here. Semantics-driven design forces us to carefully consider the semantic domain at the start, after which it remains relatively fixed while we incrementally extend the language with new syntax—this is exactly the strength of a shallow embedding. Syntactic flexibility is especially important during the early phases of semantics-driven design, while the language is evolving rapidly. Once the syntax is relatively stable, if a deep embedding is desired, it can be obtained from a shallow embedding by identifying a minimal set of core syntactic constructs (operations implemented as functions), replacing these by a corresponding data type `L`, and merging the original function bodies into a new function implementing the valuation from `L` to `D`. The remaining, non-core syntax functions remain as syntactic sugar that produce values of the abstract syntax `L`.

THE SEMANTICS-DRIVEN DESIGN PROCESS

In this section we will describe the semantics-driven design process in some detail. We will illustrate and discuss each step through the incremental development of a calendar DSL.

The semantics-driven design process consists of two major parts. The first part is concerned with the modeling of the semantic domain, which is based on the identification of basic semantic objects and their relationships. The second part consists of the design of the language's syntax, which is about finding good ways of constructing and combining elements of the semantic domain. Before delving into the details of semantics-driven design, however, we provide a high-level overview of the entire process. This will allow us to explain how concepts in the three involved realms—domain, language, and metalanguage—are related and combine to facilitate the semantics-driven design process.

---

[2] Another strategy that is available in Haskell specifically is to overload functions using Haskell's type classes to produce different semantics (Carette et al., 2009).

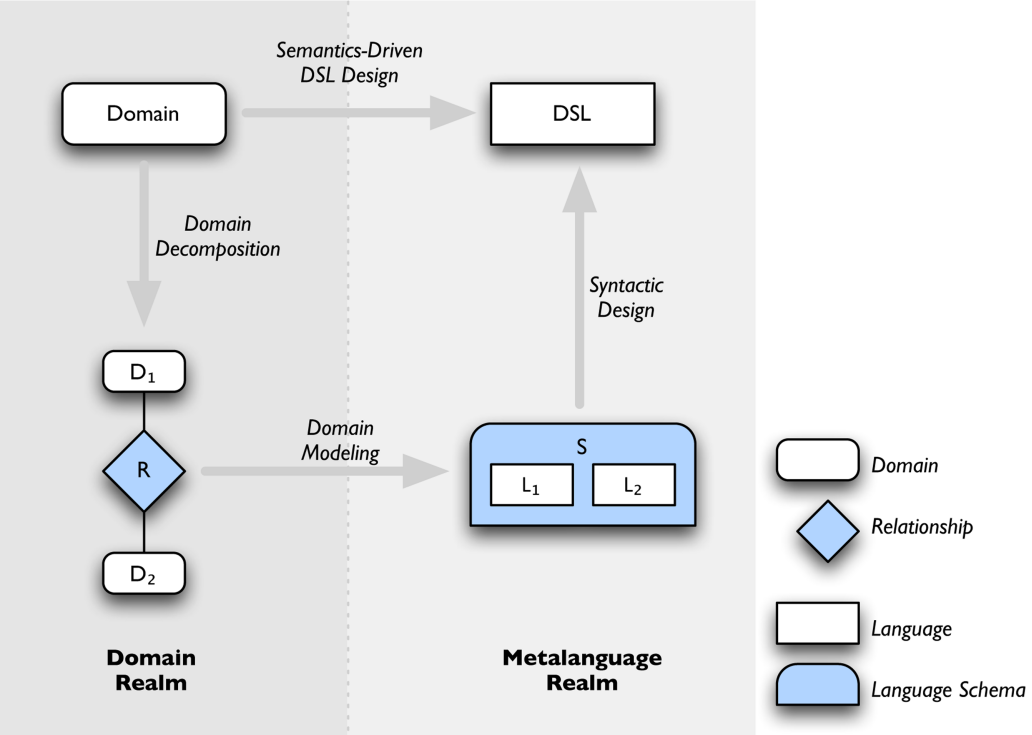

Figure 2: *Schematic illustration of the steps in the semantics-driven design process and their relationships. The two steps "Domain Decomposition" and "Domain Modeling" taken together comprise the Semantic Modeling part of the design process. The Syntactic Design step can be further distinguished as Inter- and Intra-DSL Syntax Design.*

Process Overview

Semantics-driven design leads from a problem domain to a domain-specific language that is described, or implemented, by a metalanguage. The process consists of three major steps, which are illustrated in Figure 2.

The first step decomposes the problem domain into smaller subdomains and identifies the relationships between them. In Figure 2 we find two subdomains D_1 and D_2 and a relationship R between them. This decomposition determines the semantic domain of our DSL. This step happens completely within the problem realm. No metalanguage concepts are invoked yet.

The second step concerns the modeling of the decomposed semantic domain in the metalanguage. Each subdomain forms the basis of (that is, is the semantic domain for) a little language called a *micro DSL*. The identified relationships between subdomains are modeled as *language schemas*. In Figure 2 we observe that each domain D_i is modeled as a micro DSL L_i and that the relationship R is modeled as language schema S. This step takes the DSL design from the problem realm into the metalanguage realm. In terms of Haskell, domain modeling means to define types to represent the semantics domains of languages and type constructors to represent the semantic domains of language schemas.

These first two steps taken together comprise the *semantic design* part of the DSL design process. All decisions regarding the semantics of the DSL happen in this part, which will be illustrated with the help of an example in the next subsection on "Semantic Design".

The third step in the design process is the design of the syntax of the DSL. This step can also be broken down into two parts. Specifically, we can distinguish between the syntactic design of each micro DSL and the design of syntax that spans several of these micro DSLs, leading to constructs that build

relationships between these elementary objects. We have not specifically illustrated the separate parts of syntactic design in Figure 2 since this happens completely within the metalanguage. We will talk about this in detail in the subsection on "Syntactic Design" and also discuss how the design of syntax is guided by the already defined semantics.

Semantic Design

At the core of semantic design is the identification of the essential objects that the language must construct, refer to, and manipulate. This process consists of two parts: identifying and decomposing the semantic domain into smaller (sub)domains, then modeling each domain as a micro DSL within the metalanguage.

In the first part, each decomposition of a domain into subdomains can be described by an equation that shows the relationship between the decomposed domain and its subdomains. That is, a domain is modeled and decomposed by a set of equations of the following form.

$$D_i = R_j(D_k,\ldots)$$

In the second part, these equations are directly translated into a set of type definitions. Each such definition forms the semantic basis for a micro DSL, and the relationships between the DSLs is captured through the application of type constructors that implement language schemas.

```
type L_i = S_i L_k ...
```

In the following we will demonstrate and explain these two parts in detail.

Domain identification and decomposition

As an illustrative example, we consider again the design of a small calendar DSL. The existence of many competing calendar tools with different feature sets reveal that this is no trivial domain. That some calendar functionality is often performed by external tools (such as scheduling meetings with Doodle) suggests room for improvement.

To identify the domains involved in the language we first ask ourselves, what are the essential objects involved? The basic purpose of a calendar is to define (and remind of) appointments at particular times. We recognize two separate components in this description, "times" and "appointments". Each of these will be a subdomain of calendars, leading to two domain models and two micro DSLs in the subsequent steps.

Decomposing the calendar domain into these subdomains involves identifying the relationship between times and appointments in calendars. In this case, we observe that times are *mapped* to appointments. This mapping from the subdomain of times to that of appointments captures the essence of calendars. The mapping relationship corresponds on the language level to a language schema. A language schema represents a whole class of related languages and can be obtained from a language by parameterizing some of its parts. We can produce a language from a language schema by substituting languages for its parameters. The language schema for mapping one domain to another obviously has two parameters, representing the domain and range of the mapping.

In Haskell, we can represent the result of the domain identification and decomposition step with a simple type definition. In the following, the types `Time` and `Appointment` represent the subdomains we identified for the calendar application, while the two-parameter type constructor `Map` represents the language schema that relates them.

```
type Calendar = Map Time Appointment
```

The type `Calendar` represents the calendar DSL and is composed of the `Time` and `Appointment` micro DSLs.

Note that we can also leave some aspects of the domain parameterized, producing a class of related DSLs that can be instantiated for different subdomains; that is, we can define our DSL itself as a language schema. For example, suppose we want to define the semantic domain of a range of calendar DSLs that can work for many different types of appointments. We partially instantiate the `Map` language schema only with the subdomain for `Time`, defining a parameterized domain for calendars. This can be implemented in the metalanguage as follows.

```
type CalT a = Map Time a
```

We can produce the original `Calendar` semantics by instantiating this schema as `CalT Appointment`.

Having identified and decomposed the domain of calendars, we recursively consider the `Time` subdomain. We quickly realize that clock time alone is not sufficient since we want to also be able to define appointments on different dates. We might choose an abstract representation of time that incorporates this information, like seconds since January 1, 1970, but this is not very evocative from a language perspective. Instead, we choose a compositional representation based on the above description—we want to define appointments in terms of a date and (clock) time, so we will decompose the domain on the left-hand side of the calendar mapping further into the subdomains of Date and Time. The relationship between these domains can be captured by a language schema with two parameters that represents pairs. In the metalanguage, we will use Haskell's special syntax `(Date,Time)` to represent the application of the pairing schema to the `Date` and `Time` subdomains, and define a new calendar language schema as follows.

```
type CalDT a = Map (Date,Time) a
```

This decomposition results in semantic values that are quite redundant, however. Whenever we schedule appointments on the same date, the corresponding `Date` value will be repeated in the semantics. As a solution, we will reconsider the decomposition of our semantic domain in order to factor out the redundancy. We do this by creating a new language schema for date-only calendars, `CalD`, that maps dates to an arbitrary domain. Then we compose the `CalD` and `CalT` schemas to produce a date calendar of time calendars; that is, a mapping from dates to a nested mapping from times to some appointment domain.

```
type CalD a = Map Date a
type Cal  a = CalD (CalT a)
```

This example demonstrates the compositional power of language schemas. We will consider `Cal` to be the primary decomposition of the calendar semantics domain going forward, although we will sometimes also refer to other calendar domains by their name in the metalanguage.

Note that we could also have chosen to compose `CalD` and `CalT` in the opposite order, producing a mapping first from time and then to date. Rather than localizing all appointments on the same date, this semantics would localize appointments at the same time (but on different dates). This would also serve to reduce redundancy, but violates the natural hierarchical structure of dates and times, and so does not seem to accurately reflect the meaning of a calendar.

Language schemas also provide a simple solution to the problem posed in the Introduction, of extending the language to incorporate a notion of appointment length. We can simply add a `Time` value to each appointment by (partially) instantiating the `Cal` domain in the following way.

```
type CalL a = Cal (a,Time)
```

This defines a calendar domain in which each appointment is a pair of some arbitrary appointment value and a time value representing the length of the appointment. This extension is modular in the sense that the change to the semantic domain is localized, allowing us to directly reuse any syntax or operations that are polymorphic in the appointment subdomain (that is, that have types like `Cal a`). Recall that in the syntax-driven language from the Introduction, this extension led to an entirely new domain representation and forced changes to existing, unrelated parts of the language definition.

As the derivation of `Cal` demonstrates, identifying the best semantic domain is typically an iterative process. Although we present the steps linearly, often domain identification interacts with domain modeling, as the precise modeling of a semantic domain can lead to new insights about its decomposition.

Domain modeling

Having identified the required domains and relationships for the calendar DSL, we can now start to model them, in detail, in our metalanguage. We begin with the `Map` language schema. Depending on the choice of metalanguage, generic language schemas might already exist in libraries.[3] Of course, if a required schema does not exist, or if we want more control over the representation, we have to define it ourselves. We employ the following definition. (Here and in the following we omit some Haskell-specific details, such as the definition of standard class instances.)

```
data Map a b = a :-> b | Map a b :&: Map a b
```

This definition provides two infix constructors to build maps, one for building individual associations (`:->`), and one for composing two maps into a bigger map (`:&:`).

Next we consider the representation of our subdomains of `Date` and `Time`. A date consists of a month and a day, so we model the `Date` domain in the following straightforward way.[4]

```
data Month = Jan | Feb | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov | Dec
type Day = Int
data Date = D Month Day
```

We model the `Time` domain in an equally obvious way.

```
type Hour = Int
type Minute = Int
data Time = T Hour Minute
```

That these definitions are so straightforward is a good thing. By simplifying the language design process through systematic decomposition, we make it less surprising and ad hoc and thus more comprehensible.

Note that we can consider both `Date` and `Time` to be composite domains. For example, we can decompose `Time` into the extremely simple domains of `Hour` and `Minute`. We combine these domains in Haskell within a data type, but if we wanted to make the decomposition explicit, we could instead use the pairing schema to model the domain as `(Hour,Minute)`. We choose the given data type representation of `Time` because it allows us to enforce syntactic constraints with the Haskell type system. That is, we can define operations that take `Time` values as arguments and ensure that they can only be applied to times and not to arbitrary pairs of integers (as the pair representation would allow). It is not uncommon for aspects of the metalanguage to influence minor design decisions in this way.

Finally, in order to instantiate the `Cal` language schema and write actual calendar programs, we need to identify and model an appointment domain. We can imagine several more or less complicated

[3] For example, we could reuse `Data.Map` from the Haskell standard libraries.

[4] We omit years from dates just for simplicity in this chapter.

representations that vary depending on the context and kinds of information we need to track. To keep the discussion focused on more interesting aspects of the design, we assume appointments are given by plain strings and write our first program in the calendar DSL as follows.

```
week52 :: Cal String
week52 = D Dec 30 :-> (T 8 0 :-> "Work") :&: D Dec 31 :-> (T 22 0 :-> "Party")
```

While we can express calendars by directly building semantic values in this way, it is not very convenient. The need to use the `D` and `T` constructors is annoying, and there is no way to directly express high-level concepts like repeating or relative appointments—entering such appointments manually is not only inconvenient but also error prone. Both of these shortcomings can be addressed by extending the DSL with new syntax.

Syntactic Design

Having realized a semantic core for our language, we now focus on building up its concrete syntax through the identification and implementation of new operations. Although we argue for designing the semantics first, this should not be misunderstood as devaluing the importance of syntax. On the contrary, the syntax of a language is extremely important. Good syntax has a significant impact on the usability of a language. Syntax can facilitate the expression of recurring patterns or templates, and be used to impose constraints on what can be expressed to help avoid erroneous programs.

Although each step of the semantics-driven design process provides feedback that may alter decisions made earlier, it is important that the design of syntax comes conceptually *after* the development of the semantics core. Adding new operations is an inherently ad hoc process since it is impossible to foresee all desired features and use cases. Building on a solid semantics core ensures that (1) these syntactic extensions are implemented in a consistent and principled way, and (2) the particular selection of operations does not fundamentally alter the expressiveness of the language. In a shallow embedding in Haskell, we can see that these two features are enforced by the fact that (1) all operations will be implemented as Haskell functions that produce values of the semantic domain, and (2) we can always build a value of the semantics domain directly, if a desired operation does not exist.

As we have seen in the previous section, given our choice of a shallow embedding in Haskell, the constructors of the data types `Map`, `Date`, and `Time`, which make up the semantic domain, serve directly as syntax in the DSL. Although these constructs are maximally expressive—they can obviously produce any calendar, date, or time captured by our semantic domain—they are also very nonspecific and low level. Through the addition of new operations, we can make the syntax more descriptive (and hence more usable and understandable) and raise the level of abstraction with new high-level operations that produce complex combinations of the low-level constructors.

One of the biggest advantages of the semantics-driven process is that it decomposes the difficult task of designing a DSL into several smaller and more manageable subproblems. At the level of semantic domains, as we saw in the previous sections, this process is completely modular: we identified the subdomains, decomposed the problem, and then tackled each one separately. We can perform a similar decomposition by subdomains at the syntactic level, developing syntax for the reusable micro DSLs for dates and times. However, we will also want syntax that spans and *integrates* multiple subdomains within the larger domain of calendars.

In the rest of this section, we will demonstrate the design and structured implementation of both syntactic levels. First, we develop the micro DSLs for dates and times. Then we develop syntax that integrates these languages into the larger calendar DSL.

Micro DSL syntax

Syntax that is specific to a particular subdomain is modular in the sense that, combined with its semantic domain, it forms a micro DSL that can be reused in other, completely unrelated languages. Therefore, throughout the syntax development process, it is important to identify which subdomains new operations affect and associate the syntax accordingly. In Haskell, our new operations are implemented as functions, so determining the affected domains is as simple as examining their types.

When working with DSLs in Haskell, one of simplest, yet often tremendously useful syntactic extensions is the introduction of so-called *smart constructors*. These are values or functions that supply constructors of a data type with some or all of their arguments. For example, in the `Date` micro DSL, we can introduce a function for each month as follows.

```
[jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec] = map D [Jan .. Dec]
```

With these smart constructors we can now build dates more conveniently. For example, we can write more shortly `dec 31` for `D Dec 31`, hiding the `D` constructor of the `Date` data type. In addition, we can introduce all kinds of functions for constructing, say lists of individual dates (for example, the federal holidays) or ranges of dates. We can define addition or subtraction of dates or, if we had the year information, a weekday predicate or filter. The decision of which operations to define depends, of course, on the requirements of the concrete application.

For the `Time` DSL we define several similar operations as Haskell smart constructors. The `hours` operation *specializes* the more general `T` constructor and is used for producing times on the hour. The `am` and `pm` operations add support for 12-hour clocks by translating their arguments into 24-hour time.

```
hours h = T h 0
am h = hours h
pm h = hours (h+12)
```

Finally, we add operations `before` and `after` that support the definition of relative times by adding and subtracting times from each other.

```
before t t' = t'-t
after  t t' = t'+t
```

We also add another specializing operation `mins` that produces a `Time` value containing only minutes, useful for computing relatives times.

```
mins m = T 0 m
```

A nice feature of Haskell as a metalanguage is that it offers a fairly high degree of syntactic flexibility for the development of DSLs. As an example of this, the above operations for expressing relative times are intended to be used infix, such as `hours 3 ‘after‘ pm 2` which produces a time corresponding to three hours after 2pm, or `T 17 0`. Ideally we would refine this even further to make the `hours`, `mins`, `am`, and `pm` operations postfix. Okasaki (2002) demonstrates how such an effect can be achieved in Haskell.

The very small amount of subdomain-specific syntax provided here for our two micro DSLs is by no means comprehensive. But since the semantics-driven approach and our shallow embedding in Haskell directly support the incremental and compositional extension of languages with new syntax, this is not a problem. We can just add new operations to the languages later, as needed. This syntax is also necessarily still quite low-level since the domains themselves are very constrained. In the next step we will encounter more complex operations by considering operations that integrate multiple semantic subdomains.

Domain integration syntax

Syntax that associates or combines multiple subdomains is necessarily less modular than the micro DSLs developed in the previous step. Domain integration syntax represents higher-level operations associated with larger DSLs that are made up of multiple subdomains. The compositional design of semantics-driven DSLs means that such syntax can also easily be developed for the solution of some particular problem.

For the purposes of this section it will be useful to separate the notion of *schedules* and *calendars*. In fact, we have already defined the corresponding semantic domain of schedules—mappings from clock times to appointments—as the `CalT` language schema. The `Cal` language schema can then be equivalently defined as a mapping from dates to schedules.

```
type Sched a = CalT a
type Cal   a = Map Date (Sched a)
```

The most fundamental example of domain integration syntax in the calendar DSL is the mapping operator `:->`. In the context of schedules, this operator integrates the domain of `Time` with the parameterized appointment domain `a`, bringing them into the domain of schedules. In the context of calendars, the operator integrates the `Date` and `Sched a` domains into calendars.

As a higher-level example, consider an operation `wakeAt` that captures the schedule of a morning routine. This is something we might want to add to a calendar regularly if we're either especially disciplined or forgetful. The argument to this operation is the time to wake up and the result is a schedule.

```
wakeAt :: Time -> Sched String
wakeAt t = t :-> "Shower" :&: mins 20 `after` t :-> "Breakfast"
```

So `wakeAt (hours 7)` would produce the following schedule.

```
T 7 0 :-> "Shower" :&: T 7 20 :-> "Breakfast"
```

The type of the `wakeAt` function reveals that the operation integrates times into schedules. We can combine this schedule with a date to produce a calendar using the `:->` operator, for example `jan 1 :-> wakeAt (pm 1)`.

Now, suppose we want to define a schedule that repeats indefinitely. Since we represent calendar domains as explicit mappings, this may seem at first impossible. But this is another case where the specifics of our metalanguage can influence the design of our DSL. Lazy evaluation in Haskell allows us to define such an operation by enumerating the infinite stream of dates and mapping each one to a schedule as follows.[5]

```
everyDay :: Date -> Sched a -> Cal a
everyDay d s = map (:-> s) [d..]
```

For example, if you join the military on October 10th, you might write:

```
oct 11 `everyDay` wakeAt (hours 4)
```

This very high-level operation demonstrates just how far you can get with a simple semantics core and a sufficiently powerful metalanguage.

So far all of our operations have concerned the construction of calendars and schedules. Of course, we can also define operations that modify them. Below is an operation `move f t` that reschedules any

[5] Note that without an associated year value the stream of dates is actually cyclical.

appointments from time `f` to time `t`. The first case propagates the move operation into subschedules, while the second changes the time of a mapping if it is scheduled for time `f`.

```
move :: Time -> Time -> Sched a -> Sched a
move f t   (l :&: r) = move f t l :&: move f t r
move f t m@(u :-> a) = if f == u then t :-> a else m
```

To demonstrate, the following DSL program defines a schedule with two events, then reschedules one of them, resulting in the new calendar `T 12 0 :-> "Lunch" :&: T 14 0 :-> "Meeting"`.

```
busyDay   = T 12 0 :-> "Lunch" :&: T 13 0 :-> "Meeting"
longLunch = move (pm 1) (pm 2) busyDay
```

Note that we could reuse in the `move` operation the syntactic keyword trick demonstrated earlier. By including additional keywords, like `from` and `to`, we might make the operation's concrete syntax more evocative.

COMPOSITIONALITY AND LANGUAGE OPERATORS

The principle of compositionality says that the meaning of a sentence or expression is given by its structure and the meaning of its parts. This idea has a long tradition, and some of its roots can be traced back to Frege's context principle (1884). The notion of compositionality can be formalized, for example, by stipulating a homomorphism between the syntax of the language and its semantic domain (Montague, 1970).

The principle of compositionality is at work twice in the semantics-driven design process. First, the *decomposition* of domains into subdomains and their relationships *assumes* a compositional structure in the domain to be modeled. Second, the *composition* of a DSL out of several micro DSLs *exploits* the compositionality in the application of language schemas.

The second instance of compositionality brings us to the notion of *language operators*. A language operator takes one or more languages and produces a new language, either through the composition of several micro DSLs, or through the incremental addition of syntax. In terms of Haskell, a DSL is represented by a set of Haskell definitions (a Haskell program), so a language operator that transforms one DSL into another becomes effectively a Haskell metaprogram. Note that language operators usually cannot be expressed directly in the metalanguage. Instead they are given by high-level descriptions of changes to the DSL representation. In our previous work (2011) we have provided descriptions of language operators as transformation patterns of Haskell programs. This helps us understand the representation of DSLs in Haskell and also illustrates the concrete steps required to do semantics-driven DSL development in Haskell. In this section we will focus on the specific role that some of these operators play in the semantics-driven language design process.

Syntactic vs. Semantic Language Operators

In general, language operators can apply either to the syntax or the semantics of the involved languages. Since the syntax of a DSL is given in our approach through data constructors and function definitions, syntactic language operators will add, remove, or change these constructs. Likewise, semantic language operators will involve adding, removing, or changing type definitions.

In the previous section we saw several examples of syntactic extension through the addition of new function definitions. We can also extend the scope of existing operations in the DSL by applying another language operator that parameterizes an existing operation. For example, we could add a new parameter for minutes to the `pm` function. Other language operators include the inverse operations of removing

function definitions to eliminate the corresponding syntax, and removing a parameter to reduce the scope of, or *specialize* an operation. The `hours` operation in the `Time` micro DSL is an example of a non-destructive specialization of the `T` operation for constructing times. Instead of removing the minutes parameter of `T` directly, we added a new function that hid it.

In general, the application of syntactic language operators, in the semantics-driven approach, is *guided* by the semantics. This is because the functions that implement specific syntax have to produce values of the semantic domain. Since, in Haskell, semantic domains are represented by types, the definition of new syntax is in some sense type directed. For example, the definition of the syntactic operation `wakeAt` has to employ the constructors of `Map` since it must build a value of type `Sched String`.

First-Order vs. Higher-Order Language Operators

In addition to the distinction between syntactic and semantic language operators, we can also distinguish between first- and higher-order language operators. A *first-order language operator* takes one or more languages and produces a new language, while a *higher-order language operator* takes other language operators as inputs or/and produces them as outputs.

First-order language operators directly change the representation of a DSL in the metalanguage. This can take quite different forms. For example, the addition of a function or data constructor extends the represented language by a new operation. Similarly, we can extend an existing language operation by adding a new argument to the function (or constructor) that represents that operation. We can also add whole languages by adding new data types; this is often a preparatory step to combine the language with others into a bigger language. And we can rename types, functions, and constructors. Each of these operators have natural inverse operations, for example, removing functions/constructors or their arguments, which amounts to the removal or restriction of the operation, respectively. We can similarly remove data types to eliminate whole micro DSLs. Renaming is its own inverse operation.

Most first-order operations can be composed to form other, more complicated language operations. For example, the merging of two languages `L` and `L'` into one involves adding (some of) the constructors from one data type, say `L'`, to the other one, `L`, plus changing the argument types of the added constructors from `L'` to `L`. Constructors in `L'` that represent operations already represented by constructors in `L` should be removed. The types of associated syntax functions must then also be changed from `L'` to `L`.

In contrast to first-order language operators that work directly on languages, a higher-order language operator takes other language operators as inputs or produces them as outputs. At this point it is important to recognize that a language schema is itself a language operator since it can produce, via instantiation, different languages. The higher-order language operators discussed below will produce and consume language schemas.

One important higher-order language operator is *language abstraction* that takes a language (or language schema) and produces a language schema by substituting a sublanguage by a parameter. In terms of Haskell this means to take a type (or type constructor), add a parameter to its definition, and replace some argument types of its constructors by this new parameter. For example, suppose we start with a calendar definition that does not use `Map` but that defines a monomorphic type with two similar constructors, fixing the argument types to `Date` and `Appointment`.

```
data Cal' = Has Date Appointment
          | Join Cal' Cal'
```

By abstracting from the `Date` and `Appointment` sublanguages represented in that type (and renaming the constructors), we can generalize the language `Cal'` into the language schema `Map`.

Dually, *language instantiation* takes a language schema and substitutes a language (or language schema) for one of its parameters, producing a language or a more specific language schema. For example,

`CalD` was obtained from `Map` by substituting `Date` for `a`, and `Cal` was obtained from `CalD` by substituting `CalT a` for `a`.

As with first-order language operations, we can derive more elaborate higher-order language operators from abstraction and instantiation. A very powerful derived language operator is the *composition* of language schemas. The basic idea behind schema composition is to instantiate one schema with another. We have seen an example of this already in the definition of `Cal` by composing `CalD` and `CalT`. As another example, consider the following language schema `Access` for distinguishing between private and public information. Private information is protected by some kind of key, which we assume for simplicity to be represented by strings.

```
type Key = String
data Access a = Protected Key a | Public a
```

We can compose the `CalD` language schema with the `Access` schema to obtain a calendar schema in which appointment information can be protected by this privacy micro DSL.

```
type CalDA a = CalD (Access a)
```

Unlike many other language operators, we can actually define an operator for language schema composition within Haskell quite easily.

```
type Compose s t a = s (t a)
```

This definition is analog to the definition of function composition, it just works on the level of types. With this language operator we can give an alternative definition for `CalDA` that reflects the explicit application of the employed language operator.

```
type CalDA a = Compose CalD Access a
```

Higher-order language operators, such as language abstraction, instantiation, and composition, support the semantics-driven DSL design process by facilitating gradual changes to the overall structure of a language. These operators make it possible to employ language refactorings during the design process, supporting the incremental and iterative design of compositional languages.

RELATED WORK

Foundational and related work in the area of programming languages and functional programming has been discussed already in the background section (and elsewhere throughout the paper). In this section we briefly describe the relationship of syntax- and semantics-driven design to other theories about language design and software engineering. We also list a few examples of successful, real-world DSLs that exhibit the characteristics of semantics-driven design.

Languages centered around concrete syntax are often based on the non-compositional LL or LR parsing frameworks, an approach that imposes inherent limits on the composition of languages (Kats et al., 2010). The syntax-driven approach to language design is also indirectly promoted by the popular strategy of *user-centered design* (Norman & Draper, 1986). Since user-centered design asks users about how to solve specific tasks, there is the danger of focusing on too many details of too specific operations and losing the big picture. A critical view of user-centered design was provided by Don Norman himself (2005) where he instead argued for *activity-centered design*. Our proposal is to go one step further and focus directly on the domain with which tasks and activities are concerned.

The motivation for semantics-driven language design is similar to that for *model-driven engineering* (MDE) (Kent, 2002; Schmidt, 2006). MDE encourages that solutions be developed first from the

perspective of the problem domain rather than from the solution domain. That is, the development of a software banking system would begin by abstractly modeling the concepts of accounts, customers, and transactions, and only then consider the translation of these concepts into a software implementation. The early emphasis on modeling attempts to manage complexity by decomposing the system into clearly defined abstractions before committing to a specific implementation. This is very similar to the early emphasis on domain identification and modeling in semantics-driven design, which leads to a more modular and compositional semantic basis before committing to a specific syntax.

Semantics-driven design is also closely related to the idea of *domain-driven design* promoted by Eric Evans (2003). An important aspect of domain-driven design is the development of a so-called *ubiquitous language*, to be used by software developers and domain experts alike. This ubiquitous language consists of terminology that closely reflects the key concepts of the domain to be modeled, which corresponds to the elements of the semantic domain in semantics-driven design. The main differences between the two approaches is that in domain-driven design, domain terminology is embedded into English when talking about language aspects, whereas semantics-driven design firmly grounds domain terminology in a metalanguage that is used to precisely and unambiguously define the semantic domain and the domain-specific language.

Finally, although in this chapter we name and provide structure to the semantics-driven design process, it reflects a philosophy that has long existed and proven very successful in the functional programming community. Many semantics-driven Haskell DSELs have found success in the real-world. Examples include the PFP (Probabilistic Functional Programming) library for representing and computing with discrete probability distributions (Erwig & Kollmansberger, 2006) and the Pan language for creating and manipulating images (Elliott, 2003). Interestingly, semantic values in Pan are not ground values but functions. Pan syntax therefore consists of functions that manipulate and produce other functions. Such DSELs are called *combinator libraries* (Wallace & Runciman, 1999) and represent a simple but powerful extension of the basic process sketched in third section. Perhaps the most successful combinator library is Parsec, a widely-used DSEL for constructing recursive-descent parsers (Leijen & Meijer, 2001). Parsec's expressiveness and extensibility have led to it being ported to at least a dozen different host languages.

CONCLUSION

We advocate shifting the attention in the early phases of DSL design toward semantics. We argue that a semantics-driven and compositional approach to design leads to better DSL designs that are more general and reusable, and less ad hoc. The language development process is supported by one's choice of meta-language. We suggest Haskell as a good metalanguage for semantics-driven DSL design since it supports a clear interpretation of semantic domains as types, enables the incremental extension of syntax through function definition, in addition to other helpful features like a flexible syntax and lazy evaluation.

A beneficial side effect of compositional language design is that it also leads to compositional languages, in particular, when compared to syntax-driven language design. Compositionality is generally a highly valued feature of languages since it supports expressiveness with few language constructs. In a sense, compositional languages are more economical since they provide more expressiveness with fewer constructs.

REFERENCES

Carette, J., Kiselyov, O., & Shan, C. C. (2009). Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *Journal of Functional Programming*, 19(5):509–543.

Elliott, C. (2003). Functional Images. In J. Gibbons & O. de Moor, editor, *The Fun of Programming*, pages 131–150. Palgrave MacMillan.

Erwig, M. & Kollmansberger, S. (2006). Probabilistic Functional Programming in Haskell. *Journal of Functional Programming*, 16(1):21–34.

Erwig, M. & Walkingshaw, E. (2011). Semantics First! Rethinking the Language Design Process. In *Int. Conf. on Software Language Engineering*. To appear.

Evans, E. (2003). Domain-Driven Design: *Tackling Complexity in the Heart of Software*. Addison-Wesley Professional.

Felleisen, M., Findler, R. B., & Flatt, M. (2009). *Semantics Engineering with PLT Redex*. MIT Press, Cambridge, MA.

Fowler, M. (2005). Language Workbenches: The Killer-App for Domain Specific Languages? `www.martinfowler.com/articles/languageWorkbench.html`.

Fowler, M. (2010). *Domain-Specific Languages*. Addison-Wesley Professional.

Frege, G. (1884). *Die Grundlagen der Arithmetik: Eine logisch-mathematische Untersuchung über den Begriff der Zahl*. Breslau.

Hudak, P. (1998). Modular Domain Specific Languages and Tools. In *IEEE Int. Conf. on Software Reuse*, pages 134–142.

Kats, L. C. L., Visser, E., & Wachsmuth, G. (2010). Pure and Declarative Syntax Definition: Paradise Lost and Regained. In *ACM Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 918–932.

Kent, S. (2002). Model Driven Engineering. In *Integrated Formal Methods*, pages 286–298.

Leijen, D. & Meijer, E. (2001). Parsec: Direct Style Monadic Parser Combinators for the Real World. Technical Report UU-CS-2001-35, Department of Information and Computing Sciences, Utrecht University.

Mernik, M., Heering, J., & Sloane, A. M. (2005). When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344.

Mitchell, J. C. (1998). *Foundations for Programming Languages*. MIT Press, Cambridge, MA.

Montague, R. (1970). Universal Grammar. Theoria, 36:373–398.

Norman, D. A. (2005). Human-Centered Design Considered Harmful. *ACM Interactions*, 12(4):14–19.

Norman, D. A. & Draper, S. W. (1986). *User-Centered System Design: New Perspectives on Human-Computer Interaction*. Erlbaum Associates.

Okasaki, C. (2002). Techniques for Embedding Postfix Languages in Haskell. In *ACM SIGPLAN Workshop on Haskell*, pages 105–113.

Peyton Jones, S. L. (2003). *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, UK.

Pfeiffer, M. & Pichler, J. (2008). A Comparison of Tool Support for Textual Domain-Specific Languages. In *OOPSLA Workshop on Domain-Specific Modeling*, pages 1–7.

Pierce, B. C. (2002). *Types and Programming Languages*. MIT Press, Cambridge, MA.

Schmidt, D. A. (1986). *Denotational Semantics*. Allyn and Bacon, Newton, MA.

Schmidt, D. C. (2006). Model-Driven Engineering. *IEEE Computer*, 39(2):25–31.

Thompson, S. (2011). *Haskell – The Craft of Functional Programming* (3rd ed.). Addison-Wesley, Harlow, England.

Walkingshaw, E. & Erwig, M. (2009). A Domain-Specific Language for Experimental Game Theory. *Journal of Functional Programming*, 19(6):645–661.

Wallace, M. & Runciman, C. (1999). Haskell and XML: Generic Combinators or Type-Based Translation? In *4th ACM Int. Conf. on Functional Programming*, pages 148–159.

KEY TERMS AND DEFINITIONS

Background

- *domain-specific embedded language (DSEL)*: A DSL defined within a metalanguage that uses metalanguage constructs directly as DSL syntax. Also called an internal DSL.
- *deep embedding*: A technique for implementing DSELs where abstract syntax is represented by a data type and mapped onto the semantic domain by a valuation function.
- *shallow embedding*: A technique for implementing DSELs where syntax is defined by functions that build semantic values directly.
- *compositionality*: The principle that an object can be defined and understood by considering its parts individually, then relating them in a systematic way. A desirable property of a language's design, its semantic domain, and the expressions it contains.

Introduced in this Chapter

- *syntax-driven design*: The traditional view that the design of a language begins by identifying its (abstract) syntax, and only later describing its semantics.
- *semantics-driven design*: A language design process that begins by identifying, decomposing, and modeling the semantic domain of a language, then systematically extending it with syntax.
- *domain decomposition*: The identification and separation of a semantic domain into its component subdomains and their relationships. Enables the domain to be modeled in a structured and modular way.
- *domain modeling*: The representation of a (decomposed) semantic domain in a metalanguage with types and data types, forming a hierarchy of micro DSLs related by language schemas.
- *language schema*: A parameterized class of related languages, from which specific languages can be derived by instantiation.
- *language operator*: An operation that produces a new language from one or more languages or language schemas (and possibly other arguments). Language operators are the mechanisms by which a language is incrementally extended and built from its component parts.

ADDITIONAL READING

Augustsson, L., Mansell, H. & Sittampalam, G. (2008). Paradise: A Two-Stage DSL Embedded in Haskell. In *ACM SIGPLAN Int. Conf. on Functional Programming*, pages 225–228.

Bauer, T. & Erwig, M. (2009). Declarative Scripting in Haskell. In *Int. Conf. on Software Language Engineering*, LNCS 5969, pages 294–313.

Bauer, T., Erwig, M., Fern, A. & Pinto, J. (2011). Adaptation-Based Programming in Haskell. In *IFIP Working Conf. on Domain-Specific Languages*, pages 1–23.

Carette, J., Kiselyov, O. & Shan, C. (2009). Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *Journal of Functional Programming*, 19(5), pages 509–543.

Claessen, K. & Hughes, J. (2003). Specification-Based Testing with QuickCheck. In J. Gibbons & O. de Moor, editor, *The Fun of Programming*, pages 17–39. Palgrave MacMillan.

Claessen, K., Sheeran, M. & Singh, S. (2003). Functional Hardware Description in Lava. In J. Gibbons & O. de Moor, editor, *The Fun of Programming*, pages 151–176. Palgrave MacMillan.

Elliott, C., Finne, S. & de Moor, O. (2003). Compiling Embedded Languages. *Journal of Functional Programming*, 13(2), pages 9–26.

Erwig, M. & Walkingshaw, E. (2009). A DSL for Explaining Probabilistic Reasoning. In *IFIP Working Conf. on Domain-Specific Languages*, LNCS 5658, pages 335–359.

Erwig, M. & Walkingshaw, E. (2009). Visual Explanations of Probabilistic Reasoning. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 23–27.

Holtzblatt, K., Burns Wendell, J. & Wood, S. (2005). *Rapid Contextual Design: a How-To Guide to Key Techniques for User-Centered Design*. Elsevier/Morgan Kaufmann, San Francisco, CA.

Hudak, P. (1996). Building Domain-Specific Embedded Languages. *ACM Computing Surveys*, 28(4es):196–196.

Hudak, P. (2003). Describing and Interpreting Music in Haskell. In J. Gibbons & O. de Moor, editor, *The Fun of Programming*, pages 61–78. Palgrave MacMillan.

Hutton, G. & Meijer, E. (1998). Monadic Parsing in Haskell. *Journal of Functional Programming*, 8(4).

Klint, P., Lammel, R. & Verhoef, C. (2005). Toward an Engineering Discipline for Grammarware. *ACM Trans. Software Engineering Methodology*, 14:331–380.

Kiselyov, O. & Shan, C. (2009). Embedded Probabilistic Programming. In *IFIP Working Conf. on Domain-Specific Languages*, LNCS 5658, pages 360–384.

Liang, S., Hudak, P. & Jones, M. (1995). Monad Transformers and Modular In- terpreters. In *ACM Symp. on Principles of Programming Languages*, pages 333–343.

Merkle, B. (2010). Textual Modeling Tools: Overview and Comparison of Language Workbenches. In *ACM Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 139–148.

Peyton Jones, S. L. & Eber, J. M. (2003). How to Write a Financial Contract. In J. Gibbons & O. de Moor, editor, *The Fun of Programming*, pages 105–129. Palgrave MacMillan.

Peyton Jones, S. L., Meijer, E. & Leijen, D. (1998). Scripting COM components in Haskell. In *Int. Conf. on Software Reuse*, pages 224–233.

Pfenning, F. & Elliott, C. (1988). Higher-Order Abstract Syntax. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 199–208.

Sheard, T. (2001). Accomplishments and Research Challenges in Meta-Programming. In *Int. Workshop on Semantics, Applications, and Implementation of Program Generation*, LNCS 2196, pages 2–44.

Tobin-Hochstadt, S., St-Amour, V., Culpepper, R., Flatt, M. & Felleisen, M. (2011). Languages as Libraries. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 132–141.

Walkingshaw, E. & Erwig, E. (2011). A DSEL for Studying and Explaining Causation. In *IFIP Working Conf. on Domain-Specific Languages*, pages 143–167.