Semantics First!
Rethinking the Language Design Process[⋆]

Martin Erwig and Eric Walkingshaw

School of EECS
Oregon State University

Abstract. The design of languages is still more of an art than an engineering discipline. Although recently tools have been put forward to support the language design process, such as language workbenches, these have mostly focused on a syntactic view of languages. While these tools are quite helpful for the development of parsers and editors, they provide little support for the underlying design of the languages. In this paper we illustrate how to support the design of languages by focusing on their semantics first. Specifically, we will show that powerful and general language operators can be employed to adapt and grow sophisticated languages out of simple semantics concepts. We use Haskell as a metalanguage and will associate generic language concepts, such as semantics domains, with Haskell-specific ones, such as data types. We do this in a way that clearly distinguishes our approach to language design from the traditional syntax-oriented one. This will reveal some unexpected correlations, such as viewing type classes as language multipliers. We illustrate the viability of our approach with several real-world examples.

1 Introduction

How do we go about designing a new language? This seems to still be an open question. To quote Martin Fowler from his latest book [6, p. 42]:

> *When people reviewed this book, they often asked for tips on creating a good design for the language. ... I'd love to have a [sic] good advice to share, but I confess I don't have a clear idea in my mind.*

This quote underlines that even though the development of software languages is supported by quite a few tools, the design process itself is still far from being an engineering discipline. Many concepts remain poorly defined or are interpreted differently depending on the approach taken.

In this paper we will address this problem and present a systematic approach to designing a software language. This approach is based on two key ideas or insights.

First, language development should be *semantics driven*, that is, we start with a semantics model of the language core and then work backwards to define a more and more complete language syntax. This is a rather unorthodox, maybe even heretical, position given the current dogma of programming language specification. For example, the very first sentence in Felleisen et al.'s latest book [5] states rather categorically:

⋆ 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.

The specification of a programming language starts with its syntax.

We challenge this view and argue that, even though the "syntax first" approach has a long and well established tradition, it is actually impeding the design of languages.

Second, language development should be *compositional*, that is, bigger languages should be composed of smaller ones using well-defined language *composition operators*. The notion of compositionality itself is widely embraced and praised as a mark of quality, particularly in the area of denotational semantics [15], and compositionality within individual languages is generally valued since it supports expressiveness with few language constructs. We will illustrate that a semantics-driven language design will itself be compositional in nature and will lead more naturally to compositional languages, in particular, when compared to syntax-driven language design. One reason might be that thinking about a language's syntax is often tied to its concrete syntax, which is problematic since the widely used LL or LR parsing frameworks are *not* compositional in general and thus impose limits on the composition of languages [11].

In order to motivate and explain our approach we will consider in Section 3 the (evolving) design of a small language for a calendar application. This example will help us establish a set of basic concepts and the corresponding terminology. This example also helps to point out some of the challenges that a language developer is faced with.

To discuss the involved technical aspects we have to express the language design in a concrete (meta)language. We use Haskell for this purpose since (1) Haskell has been successfully employed in the development of many DSLs, and (2) many of Haskell's concepts have a direct and clear interpretation in terms of language design. We will briefly summarize how Haskell abstractions map to language concepts in Section 2.

Using Haskell as a metalanguage (or DSL) for language design also allows us to identify new concepts in language design. One example is the notion of *language schema*. Language schemas and their cousins *language families* will be discussed in Section 4 where we specifically point out how polymorphism in the language description formalism (that is, the metalanguage) can be exploited for making language design more systematic and amenable to reuse.

A critical aspect of semantics-driven language design is the systematic, incremental extension of a base language (or schema) to a more complex language. This process is supported by language operators that are discussed in Section 5. As explained in Section 2, the semantics-driven approach leads to a distribution of language descriptions across different concepts of the metalanguage. This suggests a distinction of language operators into different categories, and the description follows these categories.

In Section 6 we will demonstrate the proposed semantics-driven language design approach on several examples to illustrate its power and simplicity. Finally, after a discussion of related work in Section 7 we present some conclusions in Section 8.

2 Haskell as a Language Design DSL

Haskell [16] has a long tradition as a metalanguage and has been used quite extensively to define all kinds of domain-specific languages. A few standard idioms of how to represent languages in Haskell have developed that are part of the Haskell folklore. Even though these idioms may not have been documented comprehensively in one place,

Language Domain		*Metalanguage (Haskell)*	
Language	L	Data type	`data L = C`^{n}
L. Schema	-	Type constructor	`data S a = C`^{n}
Program	$p \in L$	Expression	`e :: L`
Operation	$N_1 \dots N_k \to L$	Constructor	`C :: N`^{k} `-> L`
Semantics domain	D	(Data) type	`data D`
Semantics	$[\![\cdot]\!] : L \to D$	Function	`sem :: L -> D`

Fig. 1. Syntax-directed view of language concepts and their representation

Tim Sheard's paper [18] is a good place to start. There has also been some work on how to make language design modular and extensible [8, 13, 18, 19].

Most of this work is focused on syntax, taking the view that a language is defined by defining its (abstract) syntax plus a mapping to some kind of semantics domain. Under this view of language, sentences expanded from a nonterminal `L` are represented in Haskell by terms that are built using constructors of a data type `L`. Each constructor represents a grammar production for `L`. The argument types of the constructor represent the nonterminals that occur on the right-hand side of the production. Constants represent terminal symbols, and constructors having basic type arguments (such as `Int`) form the link to the lexical syntax. This view is briefly summarized in Figure 1.

The two-level view that results from the syntax-directed approach to language design is not the only way in which language can be represented, however. Alternatively, we can start the design of a language with a decision about the semantics domain that best captures the essence of the domain the language is describing. In Haskell this domain will also be represented as a (data) type, say `D`. However, its constructors will be taken immediately as language operators. For example, in a language for representing dates (see also Figure 3) we may have a data type `Month` that includes constructors such as `Jan`. This constructor would not just be considered a semantics value, but would also be used as an operation of the date language.

Of course, the language will need more operations than just the constructors of `D`. Instead of introducing an explicit representation through additional data types or constructors, as in the syntax-directed approach, the semantics-directed approach will simply define functions that take the appropriate arguments and construct elements of the semantics domain `D` directly. This idea is also behind the combinator library approach to the design of domain-specific embedded languages (DSELs), see, for example, [8].

While a semantics function is required in the syntactic approach to map explicit syntax to semantics values, in the semantics-directed approach this semantics function is effectively distributed over many individual function definitions. By forgoing an explicit syntax representation, the phase distinction between syntax and semantics disappears in the semantics-directed approach.

The semantics-directed view of language development and its implication on the metalanguage representation are briefly summarized in Figure 2.

The basic idea of semantics-driven language development is to start with a small language that represents the essence of the language to be developed, then to extend this core systematically. The advantages of this approach are: (1) The compositional

Language Domain		*Metalanguage (Haskell)*	
Language	L	Data type & functions	`data D = ` $\mathtt{C}^n$; `(f = e)`^{m}
L. Schema	-	Type constructor & functions	`data S a = ` $\mathtt{C}^n$; `(f = e)`^{m}
Operation	$N_1 \ldots N_k \to L$	Constructor or function	`C/f :: ` $\mathtt{N}^k$ ` -> L`
Semantics domain	D	Given by `D`, the data type part of L	
Semantics	$[\![\cdot]\!] : L \to D$	Given by `(f = e)`^{m}, the function part of L	

Fig. 2. Semantics-directed view of language concepts and their representation

design clearly represents the different components of the language and how they are connected, (2) the language development is less ad hoc, and (3) being compositional, the language design can be better maintained.

These advantages are particularly relevant for language prototyping. Once a designer is happy with a semantics-driven language design, she can always "freeze" it by converting its syntax from implicit into explicit representations and adding a semantics function. This enables the abstract syntax of programs in the language to be manipulated directly (for analysis or transformation), regaining any advantages of syntax-directed approaches. Therefore the semantics-driven approach should not be seen as an exclusive alternative to the syntax-directed approach, but rather as a companion that can in some cases replace it and in other cases precede it and pave the way for a more syntax-focused design.

3 Semantics-Driven Language Development

In this section we will illustrate with an example how a focus on semantics can go a long way in building a language. Specifically, we will consider compositional language extensions, that is, extensions that do not require changes to existing languages, in Section 3.1. We will also discuss the problem and necessity of non-compositional language extensions in Section 3.2.

3.1 Compositional Language Extensions

Consider a calendar tool for storing appointment information. That this is by no means a trivial application domain can be seen by the fact that many different calendar applications and tools exist with different sets of features, and that some calendar functionality is commonly performed by separate, external tools (consider, for example, the planning of schedules with Doodle).

In order to build a calendar application, we must identify the operations that such an application must perform. However, instead of doing this directly, which would lead to a flat, monolithic set of operations, we instead approach the problem by focusing first on the core language elements. We will begin by identifying the individual components of the domain, and how these can be represented by different DSLs. We will then incrementally compose and extend these smaller DSLs to form the desired application. This approach has the additional advantage that it can produce a library of small and medium-sized DSLs that can be reused in the development of many language projects. We also refer to these small, reusable DSLs sometimes as *Micro DSLs*.

```
data Month = Jan | Feb | ... | Dec
type Day = Int
data Date = D Month Day

[jan,...,dec] = map D [Jan,...,Dec]
```

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

hours h = T h 0
am h = hours h
pm h = hours (h+12)
before t t' =  t'-t
```

Fig. 3. Micro DSLs for date and time. The function `hours` constructs hour values, `am` and `pm` denote morning and afternoon hours, and the function `before` subtracts two time values.

The most important element of semantics-driven language design is the idea to start the design process by identifying the absolutely most essential concepts of the language to be developed. Compared with a denotational semantics approach, this basically amounts to identifying the semantics domain, which also means that the semantics function for this initial core language will then be the identity function.

At its core, a calendar application offers the ability to define appointments at particular times. We recognize two separate components in this description, "times" and "appointments", that we can try to define through individual DSLs. These two languages are linked to form a calendar, and without needing to know any details about times and appointments, we can see that it is this language combination that captures the essence of calendars. Specifically, times are *mapped* to appointments. We can represent this using a generic language combinator for maps that is parameterized by types for its domain and range. This straightforward `Map` data type can be defined as follows.[1]

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

Whereas a data type represents a language, a parameterized data type represents what we call a *language schema*, that is, a whole class of languages. A language can be obtained from a language schema through instantiation, that is, by substituting languages for the type parameters.

To make our example more concrete we start with a simple version of a calendar that maps days, given by month and day, to values of some arbitrary type. To this end we make use of the definitions for dates and times shown in Figure 3.[2] Note that even these tiny languages are not completely defined by data types alone. For example, functions, such as `dec` or `pm` are providing syntactic sugar. In addition, the function `before` extends the `Time` language by a new operation.

Based on the languages `Date` and `Time` we can define a language for calendars that associates appointment information with dates.

```
type CalD a = Map Date a

week52 :: CalD String
week52 = dec 30 :-> "Work" :&: dec 31 :-> "Party"
```

[1] Here and in the following we will sometimes omit details, such as `Show` instance definitions and infix declarations. Their effect will become clear from their use.

[2] Again, we simplify the definitions a bit and omit some definitions, such as the `Show` instances or the `Num` instance for `Time`.

Strictly speaking, `CalD` is still a language schema since the appointment information has not yet been fixed, but `week52` is a program in the language `CalD String`. It is obvious that we can also define a calendar language that maps `Time` to appointments. In the following example we define a simple *calendar pattern* to encode a habit to exercise before a party. With that, we can then define two typical daily schedules.

```
type CalT a = Map Time a

partyAt :: Hour -> CalT String
partyAt h = hours 2 ‘before‘ h :-> "Exercise" :&: h :-> "Party"

work, party :: CalT String
work = am 8 :-> "Work" :&: pm 6 :-> "Dinner"
party = work :&: partyAt 9
```

The use of calendar patterns supports a very high-level and compositional description of calendars without changing the underlying language representation. For example, a `party` day expands to a time calendar value as follows.[3]

```
08:00 -> "Work" & 18:00 -> "Dinner" & 19:00 -> "Exercise" & 21:00 -> "Party"
```

We argue that this form of low-cost extensibility is, at least in part, a direct consequence of choosing the most appropriate semantics domain, in this case a mapping. Therefore, the initial focus on language semantics pays off since it simplifies the later language design by dramatically lowering the language maintenance effort. Since in our view a language is given by the core representation plus additional functions, we can also view function definitions, such as `partyAt`, as user-defined extensions of the DSL.

From a language engineering perspective, we can observe that the function definition capability of the metalanguage helps us to easily and flexibly extend the core representation by new features, such as patterns for dependent appointments.

We can easily combine these two types of calendars by instantiating the date calendar with the time calendar.

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

week52 :: Cal String
week52 = dec 30 :-> work :&: dec 31 :-> party
```

We have illustrated how to extend the calendar language by refining the *domain* of the mapping structure that forms its semantics basis. In the same way we can create more expressive calendar languages by extending the *range*. Using product types we can combine appointment information with information about participants, how long an appointment takes, dependencies between appointments or other relationships, etc. This is all quite straightforward, and the result can be as general as the requirements of a particular application need it to be. This extensibility is a consequence of finding the right semantics domain for calendars at the beginning of the language design process.

[3] Note that when pretty printed, the data constructors `:&:` and `:->` are rendered as `&` and `->`.

Of course, there are situations when the initial design decision is not general enough to support a specific language extension. In these cases, we have to resort to non-compositional changes to the language. This is what we look into next.

3.2 Non-compositional Language Extensions

Suppose we want to extend the calendar application by allowing the distinction between publicly visible and private (parts of) appointments (we might, for example, want to hide the fact that we have two parties on two consecutive days). This idea can be easily extended to more sophisticated forms of visibility or visibility in particular contexts. From a language perspective we are faced with the need to selectively annotate parts of an abstract syntax tree. Since this situation is quite common, the approach to take from a language composition perspective is to define a generic annotation language (that is, a language schema) and integrate this in some way with the language schema `Cal`. We begin by defining a simple language schema for marking terms as private. This could be easily generalized to a more general annotation language by additionally parameterizing over the annotation language, but we will pursue this less-general approach for clarity.

```
data Privacy k a = Hidden k a | Public a
```

A simple extension of the calendar language with this privacy language is obtained by *composing* the language schemas `Map` and `Privacy k` (for some language of keys `k`) when instantiating `Cal`. Since language schemas are represented by type constructors in the metalanguage, language composition is realized by type instantiation.

```
type Key = String
type Private a = Privacy Key a
type CalP a = Map (Private Date) (Private a)
```

We also add some special syntax for the map constructor for different combinations of hidden and visible information. We use `*` and `.` in the smart constructors to indicate the position of the hidden and publicly visible information, respectively. (We omit the definition of `*->*` since we don't need it for our examples.)

```
(*->.) :: (Key,Date) -> a -> CalP a
(k,d) *->. i = Hidden k d :-> Public i

(.->*) :: Date -> (Key,a) -> CalP a
d .->* (k,i) = Public d :-> Hidden k i

(.->.) :: Date -> a -> CalP a
d .->. i = Public d :-> Public i
```

We can now hide data and/or appointment information in calendars (for example, to hide our birthday on New Year's Eve or that we have a party on December 30th).

```
week52 = ("pwd",dec 30) *->. "Party" :&: dec 31 .->* ("pwd","Birthday")
```

When we inspect a partially hidden calendar, the pretty printer definition for `Privacy` ensures that hidden parts will be blocked out.

```
*** -> "Party" & Dec-31 -> ***
```

So far the privacy extension of calendars was compositional. However the extension is limited. While the shown definition enables us to selectively hide information about particular appointments, it does *not* allow us to hide whole sub-calendars. This could be important because we might not want to expose the number of appointments of some part of our calendar to an outside party, but with the current definition we can only hide the leaves of the syntax tree, and the number of entries remains visible.

Note that simply wrapping `Private` around `CalP` doesn't solve this problem, because the `:&:` operation expects arguments of type `Map` and thus can't be used to compose private calendars. One could envision the definition of a smart constructor for `Map`, a function that inspects the calendar arguments and then propagates the privacy status to the combined calendar, but this approach will inherently lose the privacy information of subcalendars and thus doesn't solve the problem.

A solution to this problem is to generalize the definition of `Map` to allow for an additional language schema as a parameter, which is then used to wrap the result of recursive occurrences of `Map` in `:&:` and the arguments of `:->`. Such a generalization of `Map` itself is not compositional, but after the generalization we have regained compositionality, which allows us to continue to keep the privacy and other micro DSLs separated.

There are different ways to realize this idea. The most obvious approach is to directly apply the type constructor representing the language schema to every occurrence of `Map`.

```
data Map w a b = w a :-> w b | w (Map w a b) :&: w (Map w a b)
```

However, this representation might cause a lot of unnecessary overhead, in particular, in cases when local calendar annotations are only sparingly used. Moreover, from a more general language maintenance perspective, this approach is often more involved since one has to change *all* recursive occurrences. This might cause more work in more complicated data types, which also complicates the adaptation of values to the new types. A less intrusive approach is to add an additional constructor to `Map` which wraps just one recursive occurrence of `Map`. This constructor can then be used on demand and thus introduces the wrapping overhead only when needed.

```
data Map w a b = w a :-> w b
               | Map w a b :&: Map w a b
               | Wrap (w (Map w a b))
```

With this definition we can apply the privacy operations not only to dates and infos, but also to whole subcalendars.

```
week1 = Wrap $ Hidden "pwd" (jan 1 .->. "Party" :&: jan 2 .->. "Rest")
```

Evaluating `week52 :&: week1` produces the following output, completely hiding `week1`.

```
*** -> "Party" & Dec-31 -> *** & ***
```

The calendar scenario demonstrates how languages can be developed in small increments, starting from a small initial semantics core. We have seen that ideally language extensions are performed in a compositional way, but that this is not always possible. In the following two sections we will first briefly discuss the notions of language schemas and language families and then analyze language operators that form the basis of our approach to grow and combine languages out of small micro DSLs.

4 Language Schemas and Families

Sets of (related) languages can be characterized by a *language schema*, that is, a parameterized data type. We have seen different forms of calendars represented in this way, and all calendars are elements of the set of languages characterized by the schema `Map`.

Language schemas facilitate the definition of quite general language operators that can work on whole classes of languages. As an example, consider the function `dom` that computes the domain in the form of a list of values for any language captured by the language schema `Map`. In the calendar language `dom` computes the times at which appointments are scheduled, whereas in a scheduling or voting application (such as Doodle), where `Map` may be used to map users to their votes or preferences, `dom` computes users that have (already) voted. We can thus see that different concepts in different languages are realized by the same polymorphic function, which is made possible since the function is tied to a language schema that can be instantiated in many different ways.

Some language schemas will be the result of instantiation from more general language schemas. We have seen several examples of this, such as `CalT`, which is an instance of `Map`, and `Cal` which is a "nested instance" obtained by instantiating `CalD` (which is already an instance) by `CalT`, which is another instance.

Language schemas capture the idea of fully parameterized, or fully polymorphic, languages, represented by parametric polymorphism in data types. The generality of language schemas is a result of the data type polymorphism.

Language families are groups of related languages and are represented by type classes. Languages are related if they have common operations (methods). An important use of type classes in compositional language design is to enforce constraints on the languages that can be used in a language schema. For example, we might say that any language `w` used in the extended `Map` schema must provide an operation `unwrap`.

Type classes fit a bit differently into the "language operator" view, as will be explained below. Type classes reveal an interesting new class of activities in language design, something that could be called *language organization*. For example, creating a type class, say `F`, does *not* create a new language directly, but it provides *new opportunities* for creating new languages. This typically happens when we make a type (that is, language) `L` an instance (that is, member) of the type class (that is, language family) `F`. In that case all the functions that are derived from the type class become automatically available for the new instance. In other words, the instantiation has added new syntax (represented by the derived functions) to the language.

5 Language Operators

In our vision of semantics-driven, compositional language development, languages live in a space in which they are connected by language operators. This structure allows a language designer to start a design with some initial language and then traverse the space by following language operators until a desired language is reached. In this section we discuss the notion of *language operators*, which transform languages into one another. Specifically, we are interested in language operators for expanding languages since the semantics-driven approach to language design builds more complex languages out of simpler ones. Therefore, we will focus on expansion operators and only briefly mention their inverse cousins for language shrinking.

***First-Order Operations**: Adding/Removing …*

In the language domain	*In the metalanguage (Haskell)*	
… (Sub)language	Data type	$\bullet\langle$`data L ps = CS`$\rangle$
… Operation	Constructor	`data L ps = CS`$\bullet$`C`
… Operation argument	Constructor argument	`data L ps = CS`$\{$`C TS`$\bullet$`T`$\}$

Higher-Order Operation

In the language domain	*In the metalanguage (Haskell)*	
Abstraction	Type parameterization	`data L ps`$\oplus$`a = `$[$`a/T`$]$`CS`
Instantiation	Type instantiation	$\oplus\langle$`type L = S T`$\rangle$
Inheritance	Type class instantiation	$\oplus\langle$`instance C L where fs`$\rangle$

Fig. 4. Semantics language operators and their representation

In the description of language operators we make use of some auxiliary notation to abbreviate different kinds of changes to a language description. Since, in the context of this paper, a language description is a Haskell program, that is, a set of Haskell type and function declarations, we basically need operations to add, remove, and change such declarations. Thus, we use $\oplus$`D` and $\ominus$`D` to indicate the addition and removal of a declaration `D` from the language description, respectively. We use $\bullet$ to denote either operation. We also use these operations in the context of declarations to add or remove parts. For example, we write `data L = CS`$\oplus$`C` to express the addition of a constructor `C` to the constructors `CS` of the data type `L`. To pick a single element in a list as a context for a transformation we enclose the element in curly brackets following the list. For example, the notation `CS`$\{$`C TS`$\oplus$`T`$\}$ says that the list of argument types `TS` of one constructor `C` in the list of constructors `CS` is extended by the type `T`.

We also make use of the traditional substitution notation $[$`N/O`$]$`D` for substituting the new item `N` for the existing old item `O` everywhere it occurs in the declaration `D`, and we abbreviate $[$`N/O`$]$`O` by $[$`N/O`$]$. Specifically, we use `D` for declarations, `CS` for lists of constructors, and `C` for individual constructors. We also employ indexing to access parts of specific definitions. For example, $\texttt{CS}_\texttt{L}$ yields the constructors of the data type `L`.

We can distinguish between first- and higher-order language operators. A *first-order language operator* takes one or more languages and produces a new language. In contrast, a *higher-order language operator* takes other language operators as inputs or produces them as outputs. Moreover, we can distinguish language maintenance operations according to the language aspect they affect, that is, whether they affect the *semantics* (representation), the *syntax*, or the *organizational structure*. We will consider first- and higher-order operations for these cases separately in the following subsections.

5.1 Semantics Language Operators

The semantics language operations and their representation in the metalanguage are summarized in Figure 4.

An example of a first-order language operator is the addition of a new operation, represented in the metalanguage by the addition of a constructor to the data type representing the language. Similarly, we can extend an existing language operation by adding a new type argument to the constructor that represents that operation. We can also add whole languages by adding new data types. This will often be a preparatory step to combine the language with others into a bigger language. All of these operations have natural inverse operations, that is, removing productions/constructors, restricting operations/constructors, and removing languages/data types.

These six first-order operations form the basis for other language operations. For example, consider the case when we have two languages `L` and `M` with different operations that are nevertheless describing the same domain. We can merge `L` and `M` into one language, say `L`, by substituting all occurrences of type `M` in the constructors of `M` by `L` and then adding those updated constructors to `L`. Since language `M` is not needed anymore after the merge, it can be removed.

```
data L = CS⊕⟨[L/M]CS_M⟩
⊖⟨data M = CS⟩
```

This is an example of an (ordered) *union* of two languages (ordered, because one language is privileged since its name is kept as a result of the union).

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. We should note at this point that a language schema is itself a language operator since it can produce, via instantiation, different languages. With this in mind, we can discuss higher-order language operations. One example is *language abstraction* that takes a language or a language schema and produces a language schema by substituting a type (or sublanguage) by a parameter. Similarly, *language instantiation* takes a language schema and substitutes a language (or language schema) for one of its parameters and thus produces a language or a more specific language schema. For example, `CalD` is obtained from `Map` by substituting `Date` for `a`.

As with first-order language operations, we can derive more sophisticated higher-order language operations from abstraction and instantiation. In the following we discuss one such example, namely *language* or *schema composition*. The basic idea behind schema composition is to instantiate one schema with another. Taking the example from Section 1 we can instantiate a new language schema as follows.

```
type CalP a = Map (Private Date) (Private a)
```

We can then use this specialized schema to instantiate further languages (or schemas).

Finally, we can describe the inheritance of operations from existing languages through the instantiation of type classes, which makes type classes a powerful weapon, because in addition to the class members, all functions that are derived from the class will be made also available for the newly instantiated language. The importance of this language operation cannot be overemphasized. It can extend the scope and expressiveness of a language dramatically with very little effort. We will present an example of this later in Section 6.1.

5.2 Syntax Language Operators

The syntax language operations and their representation in the metalanguage are rather straightforward and are summarized in Figure 5. Interestingly, the syntax level offers

***First-Order Operations**: Adding/Removing ...*

In the language domain	*In the metalanguage (Haskell)*	
... Operation	Function	•⟨fun f vs = e⟩
... Operation argument	Function argument	fun f vs•v = [v/e']e
... Specialized syntax	Function instantiation	•⟨g = f e⟩

Fig. 5. Syntax language operators and their representation

only first-order language operations. This might be a reason why the semantics-driven approach is so much more powerful, because it offers higher-order language operations.

Extending a language by introducing new syntax works essentially by adding a new function definition. We have shown examples of this in Figure 3. In addition, in DSEL settings, users can extend language syntax on the fly by adding their own function definitions, as was illustrated in Section 3.1 with the function `partyAt`.

By extending an existing function with a new parameter we can extend the scope of existing operations within a language. For example, we could add a new parameter for minutes to the `pm` function shown in Figure 3 and thus extend the time language. Of course, the inverse operations of removing function definitions or removing function arguments are also available. Moreover, we can add or remove specialized syntax by adding instances of functions obtained through application of more generic functions to specific values. The definition of the reusable calendars `work` and `party` are examples of this, again happening on the user level.

5.3 Organizational Language Operators

The adjective "organizational" indicates that the operators in this group are not directly responsible for extending languages. But that does not mean that they are not useful or even powerless. Organizational operations are preparatory in nature; they are akin to an investment that pays dividend later.

For example, the definition of a type class creates a view of a language, called *language family*, that other languages can be associated with. The benefit of making a language a member of a language family (that is, making the data type an instance of the type class that represents the language family) lies in getting immediate access to all the functions that are derived from the class, that is, in language terms, the syntax of the new family member is at once expanded by the whole "family heritage". An example of this is making a language a member of `Monad`, which expands the language's syntax through all the functions available in the vast monad libraries.

The definition of a language family itself amounts to the definition of a "language multiplier" since the syntax provided by the functions derived from the type class can be repeatedly added to arbitrarily many other languages. Multi-parameter type classes, functional dependencies, and associated types do not change this view in any substantial way. Moreover, most of the machinery that is available for defining type classes, such as subclasses or derived classes, are supporting tools for the definition of language multipliers. Finally, adding a class constraint to a schema/function restricts the languages that that schema can be instantiated with. Adding a class constraint might be considered a higher-order operation since it produces a new (constrained) schema.

6 Semantics-Driven Language Design in Action

The semantics-driven approach to language development is born from our experiences designing many languages for a wide range of application domains. In this section, we discuss the design of just three of these languages from the perspective of semantics-driven design. Each of these languages is described in published papers (one with a best paper award), and one is in active use by other people. The first two strongly exhibit semantics-driven traits as published, while the third language is a more traditional syntax-directed design which we have redesigned here in a semantics-driven way.

It is important to emphasize, however, that the goal of this work is not to provide a fool-proof methodology for language engineering. Rather, it is to provide a strategy for *language design* and a toolbox for implementing this strategy. This will be evident in the following discussion, where a semantics-driven approach does not lead inevitably to an objectively best language, but rather informs design decisions and guides the inherently subjective design process.

6.1 Probabilistic Functional Programming

The first language we consider is a Haskell DSEL for probabilistic modeling, called PFP (probabilistic functional programming) [1]. This language is presented with only minor changes from the published version, made to simplify the discussion.

We begin by considering what a probabilistic model represents at a fundamental level. One obvious answer is a distribution of possible outcomes. By limiting the focus in PFP to discrete probability distributions, we can capture this meaning as a mapping from outcomes to the probabilities that those outcomes occur. We thus begin the design of PFP by partially instantiating the `Map` language schema from Section 3, creating a new language schema `Dist` for representing probability distributions.

```
type Dist a = Map a Float
```

Although we have fixed the representation of probabilities to the language of floating point numbers, this is not the only possibility; for example, probabilities might instead be represented as rational numbers.

We can now instantiate the `Dist` schema with different outcome languages to produce different distribution languages. For example, given the following simple language for coin flip outcomes, `Dist Coin` is the language of distributions of a single coin flip.

```
data Coin = H | T
```

Using this we can define distributions modeling both fair and unfair coins.

```
fair, unfair :: Dist Coin
fair   = H :-> 0.5 :&: T :-> 0.5
unfair = H :-> 0.8 :&: T :-> 0.2
```

On top of this tiny semantic core, PFP provides a large suite of syntactic extensions—operations for extending and manipulating distributions, implemented as functions. Probability distributions have several non-syntactic constraints related to probabilistic axioms. For example, probabilities in a distribution must sum to one and each be between zero and one. Operations must therefore be careful to preserve these properties.

Below we demonstrate a simple syntactic extension of the language with an operation for defining uniform distributions.

```
uniform :: [a] -> Dist a
uniform as = foldr1 (:&:) [a :-> (1/n) | a <- as]
             where n = fromIntegral (length as)
```

Using this, we could instead define the fair coin above as `uniform [H,T]`, or define the distribution of a die roll as `uniform [1..6]`. In the definition of `uniform`, we manually ensure that the probabilistic axioms are preserved and this is not too onerous. For more interesting operations that involve the composition of multiple distributions, this becomes more complicated and thus error-prone. Fortunately, organizational language operators provide a more general solution to this problem.

By observing that probability distributions form a monad, we can carefully define one composition operator (monadic bind) that preserves the axioms, along with an operator for building trivial distributions (monadic return), in order to bring distributions into the monad language family. This gives us immediate access to a huge number of monadic operations for composing and manipulating probability distributions that automatically preserve the probabilistic axioms by virtue of being defined in terms of return and bind. Interestingly, as a class of type constructors, monads are actually a family of *language schemas*. We instantiate the monad schema family for `Dist` as follows, where `toList` is a function that transforms a map, `Map a b`, into an association list, `[(a,b)]`.[4]

```
instance Monad Dist where
  return a = a :-> 1
  d >>= f = foldr1 (:&:) [b :-> (p*q) | (a,p) <- toList d
                                      , (b,q) <- toList (f a)]
```

An interesting feature of the monad language family, when using Haskell as a metalanguage, is that instantiating it also extends the concrete syntax of our language by allowing us to use Haskell's do-notation.

Now we can, for example, write the Cartesian product of two distributions by reusing the `liftM2` composition operator from Haskell's standard libraries.

```
prod :: Dist a -> Dist b -> Dist (a,b)
prod = liftM2 (\a b -> (a,b))
```

And we can confirm that the probabilistic axioms are preserved by examining the product distribution of our fair and unfair coins from above.

```
> prod fair unfair
(H,H) -> 0.4 & (H,T) -> 0.1 & (T,H) -> 0.4 & (T,T) -> 0.1
```

This demonstrates the power of language families for enabling language reuse and promoting structured language extension.

[4] Note that the following is not strictly Haskell code since we cannot instantiate a type class with a partially applied type synonym. In fact, `Dist` is a newtype, but wrapping and unwrapping the nested `Map` value uninterestingly obfuscates the code, so we ignore this detail.

However, not all operations on probability distributions can be implemented in terms of bind. One such example is computing conditional probability distributions. Given a distribution `d` and a predicate `p` on outcomes in `d`, a conditional distribution `d'` is the distribution of outcomes in `d` given that `p` is true. In other words, `p` acts as a filter on `d`, and the probabilities are scaled in `d'` to preserve the probabilistic axioms. We extend the syntax of PFP with a filter operation for computing conditional distributions, described by the following type definition.

```
(|||) :: Dist a -> (a -> Bool) -> Dist a
```

To demonstrate the use of this operator, we also define the following simple predicate on tuples, which returns true if either element in the tuple equals the parameter.

```
oneIs :: Eq a => a -> (a,a) -> Bool
oneIs a (x,y) = a == x || a == y
```

Now we can, for example, compute the distribution of two fair coin tosses, given that one of the tosses comes up heads.

```
> prod fair fair ||| oneIs H
(H,H) -> 0.33 & (H,T) -> 0.33 & (T,H) -> 0.33
```

This discussion has barely scratched the surface of PFP. In addition to many more syntactic extensions (operations on distributions), PFP provides semantic extensions for describing sequences of probabilistic state transitions, running probabilistic simulations, and transforming distributions into random (impure) events. The high extensibility of the language, both syntactically and semantically, is a testament to the benefits of semantics-driven design and an emphasis on language composition. This is also demonstrated in the next subsection, where we directly reuse PFP as a sublanguage in a larger language for explaining probabilistic reasoning.

6.2 Explaining Probabilistic Reasoning

The language described in this subsection focuses on *explaining* problems that require probabilistic reasoning [2, 3]. This language has also been simplified from previously published versions, both for presentation purposes, and to better demonstrate the semantics-driven approach.

We motivate this language with the following riddle: "Given that a family with two children has a boy, what is the probability that the other child is a girl?" Many reply that the probability is one-half, but in fact, it is two-thirds. This solution follows directly from the conditional probability example above. If a birth corresponds to a fair coin flip where heads is a boy and tails is a girl, then we see in the resulting conditional distribution that two out of the three of the remaining outcomes have a girl, and their probabilities sum to two-thirds.

Following the semantics-driven approach, the first step in designing an explanation language is to identify just what an explanation is, on a fundamental level. It turns out that this is an active area of research and hotly-debated topic by philosophers [7]. Ultimately, we opted for a simple and pragmatic explanation representation based on a story-telling metaphor, where an explanation is a sequence of steps that guide the reader from some initial state to the explanandum (that is, the thing that is to be explained).

An initial attempt to represent this semantics in Haskell follows, where the sublanguages `s` and `a` represent the current state at a step and the annotation describing that step, respectively.

```
data Step s a = Step s a
type Expl s a = [Step s a]
```

For explaining probabilistic reasoning problems, we can instantiate these schemas with a probability distribution for `s`, and a simple string describing the step for `a`.

```
type ProbExpl b = Expl (Dist b) String
```

The state of each (non-initial) step in an explanation is derived from the previous step. Rather than encode this relationship in each explanation-building operation, we instead reuse the `Step` schema to extend the semantics with a notion of a *story*. A story is a sequence of annotated steps, where each step is a transformation from the state produced by the previous step to a new state. We can then instantiate a story into an explanation by applying it to an initial state.

```
type Story s a = [Step (s -> s) a]
explain :: Story s a -> s -> Expl s a
```

As an example, we can define the story in the above riddle by a sequence of three steps: add the first child to the distribution, add the second child to the distribution, filter the distribution to include only those families with a boy. We can then instantiate this story with the empty distribution to produce an explanation—essentially a derivation of the conditional distribution from the previous subsection.

However, this explanation is somewhat inadequate since it requires the reader to still identify which outcomes in the final distribution are relevant and add up their probabilities. As a solution, we extend the semantics of probabilistic reasoning explanations by wrapping distributions in a construct that controls how they are viewed, allowing us to group together those cases that correspond to the solution of the riddle. The language schema `G` describes optionally grouped distributions. If a distribution is grouped, a partitioning function maps each element into a group number.

```
data G a = Grouped (a -> Int) (Dist a) | Flat (Dist a)
type ProbExplG b = Expl (G b) String
```

This extension is similar to the addition of privacy to the calendar language in Section 3, in that we extend the semantics by wrapping an existing sublanguage in a language schema that gives us additional control over that language. Finally, we add a fourth step to our story that groups results into two cases depending on whether the other child is a girl or boy, so the riddle's solution can be seen directly in the final grouped distribution.

In addition to several syntactic extensions for creating explanations, in [2] we also extend the semantics to include story and explanation branching, for example, to represent decision points. In [3] we provide several operations for automatically transforming explanations into alternative, equivalent explanations (which might then help a reader who does not understand the initial explanation). This extension highlights a strength of the semantics-driven approach. By focusing on a simple, fundamental representation of explanations these transformations were easy to identify, while they would have been much more difficult to extract from the (quite complex) syntax of explanation creation.

6.3 Choice Calculus

The final language we will consider is the *choice calculus*, a DSL for representing variation in software and other structured artifacts [4]. As published, this is a more traditional language, with a clear separation of syntax and semantics connected by a semantics function. Although the initial design was strongly motivated by a consideration of the semantics, we present a significantly re-designed version of the language here, using a more purely semantics-driven approach.

The essence of a variational artifact is once again a mapping. The range of this mapping is the set of plain artifacts encoded in the variational artifact (that is, its variants), and the domain is the set of decisions that produce those variants. We instantiate the `Map` schema with a language for decisions, defined below, to produce a language schema `V` (which stands for "variational") for the semantics of choice calculus expressions.

```
type V a = Map Decision a
```

For the discussion here, we will use the lambda calculus as our artifact language, represented by the following data type.

```
data LC = Var Name | Abs Name LC | App LC LC
type VLC = V LC
```

We read the `V` schema as *variational*, so `VLC` is the *variational lambda calculus*.

The best representation for decisions is not immediately obvious. One option is to employ a "tagging" approach, where each alternative in a choice (a variation point in the artifact) is labeled with a tag. A decision is then just a list of tags, one selected from each choice. This approach is appealingly simple, but turns out to be too unstructured. As a solution, we introduce in [4] locally scoped *dimensions*, which bind and synchronize related choices. For example, a dimension *OS* might include the tags *Linux*, *Mac*, and *Windows*; every choice in the *OS* dimension must then also contain three alternatives, and the selection of alternatives from these choices would be synchronized.

Therefore, we define decisions to be a list of dimension-tag pairs, representing the tag chosen from each dimension in the variational artifact.

```
type Dim = String
type Tag = String
type Decision = [(Dim,Tag)]
```

To implement locally scoped dimensions, we will need to "lift" the semantics to parameterize it with a notion of context. A context is propagated downward from selections in dimensions, so we represent it as a list associating dimensions with integers, where the integer represents the alternative to select from each choice bound by that dimension.

```
type Context = [(Dim,Int)]
```

We express the lifted semantics below and provide a function to "unlift" the semantics of a top-level variation artifact by applying an empty context.

```
type V' a = Context -> V a
type VLC' = V' LC

unlift :: V' a -> V a
unlift = ($ [])
```

Now we can define the syntax of the choice calculus in terms of this lifted semantics. We must define two operations, for declaring dimensions and introducing choices. The dimension declaration operation takes as arguments a dimension name, its list of tags, and the scope of the declaration. As before, toList transforms a Map into an association list; we introduce fromList to perform the inverse operation.

```
dim :: Dim -> [Tag] -> V' a -> V' a
dim d ts f c = fromList [((d,t):qs,e') | (t,i)   <- zip ts [0..]
                                        , (qs,e') <- toList (f ((d,i):c))]
```

The semantics of this operation is computed by selecting each tag independently in the scope (by prepending (d,i) to the context, where the selected tag is the ith tag in ts) and prepending that selection to the decision of the result.

The operation for introducing choices is much simpler. It accepts its binding dimension name and a list of alternatives as arguments, looks up its dimension in its associated context, and returns the ith alternative if an entry in the context is found.

```
chc :: Dim -> [V' a] -> V' a
chc d as c = case lookup d c of
                 Just i  -> (as !! i) c
                 Nothing -> error ("Unbound choice: " ++ d)
```

We also extend the syntax with smart constructors for variational lambda calculus expressions. Their implementations are omitted for lack of space, but each propagates the corresponding LC constructor over the argument semantics. For app, the result is a product of the two mappings, where entries are joined by concatenating the decisions and composing the resulting lambda calculus expressions with the App constructor.

```
var :: Name -> VLC'
abs :: Name -> VLC' -> VLC'
app :: VLC' -> VLC' -> VLC'
```

Finally, we provide an example of the language in action below. Note that we pretty print the dimension-qualified tag ("D","t") as D.t for readability.

```
> unlift $ dim "A" ["t","u"]
         $ app (chc "A" [var "f", var "g"])
               (chc "A" [var "x", dim "B" ["v","w"]
                                 $ chc "B" [var "y", var "z"]])
[A.t] -> f x & [A.u,B.v] -> g y & [A.u,B.w] -> g z
```

Observe that the two choices in the A dimension are synchronized and that a selection in dimension B is only required if we select its alternative by selecting A.u.

This example demonstrates the flexibility of the semantics-driven approach by showing that it can relatively easily accommodate concepts like scoping that seem at first to be purely syntactic in nature. In particular, the strategy of lifting a semantics language into a functional representation is potentially very powerful, although a full exploration of this idea is left to future work.

7 Related Work

There is a vast literature on language design that approaches the problem from a syntactic point of view; Klint et al. provide a comprehensive overview [12]. Also, the recent flurry of work on language workbenches—essentially integrated development environments that support the creation of DSLs, takes a predominantly syntax-focused approach to language design; for overviews see [17, 20, 14]. In contrast, the approach described in this paper is characterized by its focus on semantics.

Paul Hudak was among the first to advocate DSELs [8, 9] (also called "internal DSLs" [6]) and the compositional approach to developing DSLs. This work has inspired many to develop DSELs for all kinds of application areas (some impressive examples can be found in this collection [10]). Our choice of Haskell as a metalanguage raises the question of how semantics-driven DSL design relates to DSELs. The answer is that the two concepts are quite independent. For example, one might define a DSEL in Haskell by first defining the abstract syntax of the language as a data type, which is decidedly syntax-driven. Similarly, the semantics-driven approach can be equally well applied to non-embedded (external) DSLs. The creation of combinator libraries (for example, in [8]) is a specific strategy for implementing DSELs in functional metalanguages that most closely resembles the semantics-driven approach. Although the design of a combinator library will not necessarily incorporate all aspects of the semantics-driven approach, combinator libraries can nonetheless be viewed as a specific realization of the semantics-driven approach in languages like Haskell.

The use of Haskell for language design and development has also been subject to research. Tim Sheard provides an overview of basic techniques and representations [18]. One particular problem that has been addressed repeatedly is the composition of languages (or language fragments). One proposal is to abstract the recursive structure of data types in a separate definition and use a fixpoint combinator on data types to tie together several languages into one mutual recursive definition [19]. Another approach is to systematically employ monad transformers to gradually extend languages by selected features [13]. Both of these proposals are quite creative and effective. However, they embrace the syntax-oriented view of languages. This is not a bad thing; on the contrary, as far as they go, these approaches provide also effective means to compose parsers for the built languages whereas the semantics-driven approach and internal DSLs have little control over syntax. On the other hand, the opportunities for language composition are rather limited when compared with the semantics-driven approach.

8 Conclusions

In this paper, we have promoted a semantics-driven approach to language development and identified a set of language operators that support the incremental extension and composition of languages in order to realize this approach. Our approach is based on a clear separation of syntax and semantics into different concepts of the chosen metalanguage Haskell, namely functions and data types, respectively.

We have illustrated our approach with several examples, including the design of non-toy languages that have been published and are in use, which demonstrates that semantics-driven language design actually works in practice.

The advantages of our approach are particularly relevant for language prototyping. And while semantics-driven language design can in some cases replace the traditional syntax-focused approach, it can also work as a supplement, to be used as tool to explore the design space before one commits to a specific design that is then implemented using the syntactic approach.

References

1. Erwig, M., Kollmansberger, S.: Probabilistic Functional Programming in Haskell. Journal of Functional Programming 16(1), 21–34 (2006)
2. Erwig, M., Walkingshaw, E.: A DSL for Explaining Probabilistic Reasoning. In: Taha, W.M. (ed.) DSL 2009. LNCS, vol. 5658, pp. 335–359. Springer, Heidelberg (2009)
3. Erwig, M., Walkingshaw, E.: Visual Explanations of Probabilistic Reasoning. In: IEEE Int. Symp. on Visual Languages and Human-Centric Computing, pp. 23–27 (2009)
4. Erwig, M., Walkingshaw, E.: The Choice Calculus: A Representation for Software Variation. ACM Transactions on Software Engineering and Methodology (2011) (to appear)
5. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex. MIT Press, Cambridge (2009)
6. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional (2010)
7. Halpern, J., Pearl, J.: Causes and Explanations: A Structural-Model Approach, Part I: Causes. British Journal of Philosophy of Science 56(4), 843–887 (2005)
8. Hudak, P.: Modular Domain Specific Languages and Tools. In: IEEE 5th Int. Conf. on Software Reuse, pp. 134–142 (1998)
9. Hudak, P.: Building Domain-Specific Embedded Languages. ACM Computing Surveys 28(4es), 196 (1996)
10. Gibbons, J., de Moor, O. (eds.): The Fun of Programming. Palgrave MacMillan (2003)
11. Kats, L.C.L., Visser, E., Wachsmuth, G.: Pure and Declarative Syntax Definition: Paradise Lost and Regained. In: ACM Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications, pp. 918–932 (2010)
12. Klint, P., Lämmel, R., Verhoef, C.: Toward an Engineering Discipline for Grammarware. ACM Trans. Softw. Eng. Methodol. 14, 331–380 (2005)
13. Liang, S., Hudak, P., Jones, M.: Monad Transformers and Modular Interpreters. In: 22nd ACM Symp. on Principles of Programming Languages, pp. 333–343 (1995)
14. Merkle, B.: Textual Modeling Tools: Overview and Comparison of Language Workbenches. In: ACM Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications, pp. 139–148 (2010)
15. Mitchell, J.C.: Concepts in Programming Languages. Cambridge University Press, Cambridge (2003)
16. Peyton Jones, S.L.: Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press, Cambridge (2003)
17. Pfeiffer, M., Pichler, J.: A Comparison of Tool Support for Textual Domain-Specific Languages. In: OOPSLA Workshop on Domain-Specific Modeling, pp. 1–7 (2008)
18. Sheard, T.: Accomplishments and Research Challenges in Meta-Programming. In: Taha, W. (ed.) SAIG 2001. LNCS, vol. 2196, pp. 2–44. Springer, Heidelberg (2001)
19. Sheard, T., Pasalic, E.: Two-Level Types and Parameterized Modules. Journal of Functional Programming 14(5), 547–587 (2004)
20. Völter, M., Visser, E.: Language Extension and Composition With Language Workbenches. In: ACM Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications, pp. 301–304 (2010)