Proceedings of the International Multiconference on
Computer Science and Information Technology pp. 635–642
ISBN 978-83-60810-22-4
ISSN 1896-7094

Developing DSLs using combinators. A design pattern

Pablo Andrés Barrientos
Universidad Nacional de La Plata,
Facultad de informática
pablo.barrientos@sol.info.unlp.edu.ar

Universidad Nacional de Quilmes
pbarrientos@unq.edu.ar

Pablo E. Martínez López
Universidad Nacional de Quilmes.
fidel@unq.edu.ar

***Abstract*—The development of domain-specific languages (DSLs) is considered by many authors as a hard task. To simplify the design of DSLs we describe a design pattern based on the combinators technique, which can also provide guidelines for previous domain analysis phase because it is based on equational reasoning over the domain knowledge.**

Combinators is a common technique from functional programming to write programs. It was used many times to implement domain-specific embedded languages (DSELs) but that implementation approach is not the only one. In this paper we present the pattern together with the underlying and basic ideas behind it. We also show benefits of using it and illustrate the use of this pattern with some examples.

***Index Terms*—DSLs, combinators, design patterns.**

I. Introduction

WHEN a programmer writes a program, he expresses a solution for a particular problem. Although that problem's domain can be limited, usually the language used to write the solution is a general-purpose language (GPL). Domain-specific languages (DSLs) are particular languages, tailored to a particular problem domain. DSLs give the programmer, and mainly the end-user with no previous knowledge about programming, the possibility to express solutions to a problem by using domain-specific notations and constructions. These constructions and notations capture the semantics from a particular bounded domain in such a way that it becomes easy to understand, write and think by expert people of that specific domain. DSLs are concise languages and mainly declarative but they can also be imperative, depending on the application domain.

In order to define a DSL, all the concepts from the domain are defined minimizing the semantic gap that exists between the problem's domain and the program, hiding the implementation details and making the self documentation of programs easy. Additionally, the amount of code which must be written is reduced, increasing productivity and decreasing maintenance costs.

Mernik et al. [13] detailed different DSL development phases which interact with each other in the development process itself. These phases are decision, analysis, design, implementation, and deployment. They identified some commonalties or patterns in each of these phases that can be applied when someone develops a DSL.

Language design is about how to model, relate, and structure the parts of a language. In the classification by Mernik et al. [13], the design patterns are characterized along two dimensions: the relationship between the DSL and existing languages, and the formal nature of the design description.

A way to design a DSL is to base it on an existing language. Mernik et al. identified three kinds of design patterns based on this idea:

- Piggyback: an existing language is partially used
- Specialization: an existing language is restricted to provide features from a problem domain
- Extension: an existing language is extended with new features that address domain concepts

On the other hand, the developer could create a DSL from scratch, with no relationship to any existing language.

The other dimension for classifying DSL design patterns is formality: they can be either informal or formal. In an informal design the specification is made in some form of natural language. A formal design would consist of a specification written using one of the available semantic definition methods [13].

The pattern we are presenting is a formal design pattern that is also related with a formal domain analysis, because it is based on equational reasoning over the domain knowledge [14]. But it is really hard to classify the pattern in the other dimension. This pattern is related to language invention, because we use the elements and concepts defined inside the pattern to design a language from scratch. On the other hand, an existing language designed using this pattern can be extended to produce a new language. Inside the language exploitation pattern, it is really hard to establish properly the exact nature of the pattern (that is, its subpattern). We need an existing language designed using the combinators pattern in order to exploit it. DSLs designed using other patterns could be hard to exploit using combinators and this could be a strong precondition. More details about extensibility will be discussed later, when we fully describe the pattern.

The pattern we are presenting is based on a very intuitive idea for programmers: a domain should be represented as combinations of subdomains, i.e., denotational semantics is compositional (e.g., the denotation of (P1 + P2) is given by combining somehow the denotations of P1 and P2). The pattern is based on combinators. Combinators are elements that take solutions of subproblems and combine them into a solution for a given problem, thus capturing very well the idea of modular programming. They can be represented as (higher-order) functions that do not refer to global variables.

The combinators technique provides solutions by providing three things:

- a basic type to represent solutions to the problem,
- a number of basic elements (called atomic solutions) representing solutions to basic instances,
- a number of ways to combine solutions to subproblems to get solutions to more complex problems.

All non-trivial solutions must be represented as combinations of smaller solutions — this is not a limitation for the application of combinators, because most sensible problems can be analyzed by separating and dividing them.

This technique has been widely used in functional programming for developing solutions for specific domains. The same underlying ideas can be applied for DSL design, independently of the way the language is implemented.

The contribution of this paper is the identification and description of a design pattern for DSLs which can be used as part of development process. This paper presents the Combinators pattern following the precise description format proposed by Gamma for object-oriented design patterns [6]. Our intention here is two-fold: to describe the pattern in a clear and widely accepted format, and to suggest the adoption of this format for DSL design patterns description in future literature.

II. Design pattern description

Name and classification

Combinators — *formal design pattern — language exploitation/invention*[1].

Intent

To allow the design of a DSL in such a way that the end-user expresses the underlying compositional structure from a domain by defining a set of combination schemes of smaller solutions.

Motivation

Consider we want to simulate a rail network and its connecting services. The end-user specifies the components of a rail network by declaring stations, rails, and connections between them. As the structure is given by composing different rails, and attaching intermediate and final stations, the combinators are a good choice. Colors for different routes are defined in the DSL to distinguish them. Finally, the user defines all different services that use the rail network, and the system is later simulated by some program.

We start defining a function to construct simple rails with no connections as $\mathsf{rail} : \mathsf{Id} \rightarrow \mathsf{Length} \rightarrow \mathsf{Rail}$. So, $\mathsf{rail\ 1\ 30.7}$, is an example of a rail with identifier 1 and length 30.7. In the same way, stations can be described by the function $\mathsf{station} : \mathsf{Name} \rightarrow \mathsf{Station}$. These functions are combinators of very simple elements of the domain (features of the element that the function returns) that construct basic elements of our interest. These kinds of functions are called *constructors.*

The connections between rails and stations are defined by the functions:

$<> : \mathsf{Rail} * \mathsf{Rail} \rightarrow \mathsf{Rail}$
describes a link between two rails.

$-| : \mathsf{Rail} \rightarrow \mathsf{Station} \rightarrow \mathsf{Rail}$
describes the link between a rail and a station.

$|- : \mathsf{Station} \rightarrow \mathsf{Rail} \rightarrow \mathsf{Rail}$
describes a connection similar to the previous one, but with the station as the first argument.

The functions we defined above, are called simply *combinators*, and they are characterized by the kind of arguments they take, which are elements of the domain.

As the reader can observe, a station can appear many times in different rail routes. The rails also can appear in different routes. To differentiate the routes with each other, the user can transform them by giving a coloring to the routes. With this purpose, we define the function $\mathsf{coloring} : \mathsf{Color} \rightarrow \mathsf{Rail} \rightarrow \mathsf{Rail}$. This kind of combinator (that takes elements of the domain and returns a different type of element) is called *transformer.* When transformations are described in the form of equations, they can be used to guide the implementation of the static semantics of the language (with static semantics we refer to those actions performed by a compiler that depend on the program being compiled; dynamic semantics represents the computations which depends on input data of the program and they must be postponed until runtime). See the sample code section to get an example.

The user can define routes by using equations. The name of the route is the right-hand side and the description is the left-hand side of the equation.

Now we change the definition of Rail from a simple element to a synonym of functions $\mathsf{Train} \rightarrow \mathsf{Time} \rightarrow \mathsf{Service}$ — i.e., rail combinators defined before are indeed higher-order functions. Services are still defined using equations. To construct Time elements, we use standard notation (e.g., 14:30) just to give simplicity and expressiveness. We finally define the train constructor $\mathsf{train} : \mathsf{Id} \rightarrow \mathsf{Capacity} \rightarrow \mathsf{Length} \rightarrow \mathsf{Train}$.

A simple example of Buenos Aires' metro rail system is given in Figure 1 — the length of rails is not precise and several stations were omitted. Infix notation is used for binary combinators. Additionally, curried versions of functions are used. The curried version of functions is obtained by replacing a structured argument by a sequence of simpler ones. For example: $\mathsf{smaller} : \mathsf{Int} \rightarrow \mathsf{Int} \rightarrow \mathsf{Int}$ is the curried version of

[1]We follow the classification proposed by Mernik et al. [13] described in section I

```
Constitucion          =  station 'Constitucion'
Independencia         =  station 'Independencia'
DiagonalNorte         =  station 'Diagonal Norte'
Retiro                =  station 'Retiro'
Catedral              =  station 'Catedral'
Callao                =  station 'Callao'
CongresoDeTucuman     =  station 'Congreso de Tucuman'

lineC  =  coloring Blue (Constitucion |− (rail 1 0.5) <> (rail 2 0.4) −| Independencia |−
              (rail 3 1.0) <> (rail 4 0.5) −| DiagonalNorte |− (rail 5 1) |− Retiro)
lineD  =  coloring Yellow (Catedral |− (rail 11 0.5) −| DiagonalNorte |− (rail 12 0.2)
              −| Callao |− (rail 13 0.4)<> (rail 14 0.9) <> (rail 15 1))
              |− CongresoDeTucuman)

serviceC1  =  lineC  (train 1 100 50)  (9:00)
serviceC2  =  lineC  (train 1 100 50)  (11:00)
serviceD   =  lineD  (train 2 43 25)   (9:15)
```

Fig. 1. Short Buenos Aires' metro system modeled using the example DSL

smaller : Int $*$ Int $\rightarrow$ Int. So, the function is written smaller 1 2, instead of smaller (1,2).

Applicability

Use the Combinators pattern, if

- The meaning of elements from the domain can be expressed in a compositional way

Structure

The curried version of functions is usually used. In Figure 2 we show the structure of the pattern.

Participants

- *Constructors* (rail, station, train)
  To implement combinators over a particular domain some basic elements must be chosen and defined. Each element of the domain is extrapolated to what constitutes a *solution*. Each solution expresses the denotational semantics of the basic elements of the domain. These elements are defined using functions called constructors.
- *Combinators* ($|-$, $-|$, $<>$)
  Combinators are defined to combine two or more elements (solutions) of the domain to construct bigger elements and get the desired solution.
- *Transformers* (coloring)
  In some cases, the elements of the domain must be modified somehow. The *transformers* of solutions are defined to this end.

Consequences

The Combinators pattern has the following advantages and consequences:

- *The code is easy to understand and reason about*. The notation is not only easy to design, but also easy to use and reason about (equational reasoning is used when defining the meaning). It is possible even for non-programmers to understand the code of the DSL because the domain semantics is captured concisely.
- *Language extension*. If we want to enrich a DSL with new features, we can define combinators that take elements from the DSL and new elements that extend the domain. The result of each combinator is a new element belonging to a new domain (the union of the previous DSL domain and the new elements for the extension). The resulting DSL has the combinators from the former DSL, new combinators and maybe transformers for its elements. Additionally, some combinators could be joined to get a new one, to obtain a clear syntax which reflects the domain. In modular design, this idea could be used making a DSL a module and adding new features from other modules by using combinators.
  Language extension seems to be simple but the new domain must be analyzed since some inherited properties from the original domain can be invalid, and new properties could emerge.
  At the end of section *Sample code* there is an example of language extension.

Implementation

There are some issues to consider when implementing this pattern.

- *Infix notation*. The binary combinators are usually more readable when using infix.
- *Curried functions*. Functions are better written using curried versions, to get a nice and more convenient notation and to reduce the use of parenthesis.
- *Language derivation from formal specification*. The combinators approach is highly amenable to formal methods, for many of the reasons already mentioned. The key point is that one can reason directly within the domain semantics, rather than within the semantics of the programming language.
  Implementation of atomic solutions as well as combinators and transformers, could be derived from their specifications following the method proposed by Hughes [9].

// use of constructors - basic solutions
([definition$_b$ =] constructor_of_simple_element$_b$)*

// use of constructors with parameters
([definition$_c$ =] constructor_of_element$_c$ parameter$_{1_c}$ ...parameter$_{n_c}$)*

// use of combinators - complex solutions
([definition$_s$ =] combinator$_s$ element$_{1_s}$ ... element$_{n_s}$)*

// use of transformers
([definition$_t$ =] transformer$_t$ element$_{1_t}$ ...element$_{n_t}$)*

Fig. 2. Structure of the DSL code when applying the pattern

Hughes made a formal specification of pretty printers for structured data, studied the algebraic properties of such combinators, and then used the results to guide the language design and implementation, based on the ideas behind combinators.

- *Implementation approaches*. DSL design patterns are independent of the implementation pattern, but in this case the resulting code looks very functional, and this could be a problem for coding it. However, it must be observed that it is just a matter of notation. The only implementation approach that seems to be problematic is the embedding of the language in a non-functional base/host language. However, we can express the semantics of the language if we can define somehow the semantics of combinators — for instance, by using the *Composite* pattern [6] in object-oriented programming languages. C++ is an example of a non-functional language that can express higher-order and polymorphic functions, including lazy evaluation [12]. For the rest of the implementation approaches, this notation should not represent a problem.
- *Functional embedded approach*. To embed the DSL using functional languages is the easiest and suitable approach for this pattern, because semantics is written with no additional costs. Functional languages provide all the (meta) constructions used in this pattern and, in many cases, they provide syntactic facilities (like data types definitions in Haskell [19]). Existing mechanisms such as function definitions or operators with user-defined syntax are used to build objects, combinators and transformers for the resulting DSL. The syntactic mechanisms of the base language are used to express the idiom of the domain. The objects defined in the DSL represent both abstract ideas and their concrete implementation.

Sample code

We present a concrete example where the pattern is applied. We give sample codes for some of the structures and elements defined. Additionally, the language is extended to see how this is achieved.

The Feature Description Language* (FDL*)[11] is the result of a new domain analysis from FDL [16]. FDL is a textual language derived from feature diagrams [3], which is a technique used in Feature Oriented Domain Analysis (FODA) [10]. It describes the composition of application domains and captures their variable parts. For more information about FDL the reader can see the original paper [16]. FDL* restates the semantics of FDL by using an algebraic model. We describe FDL* constructions from a combinators point of view. The result is almost the same as the one obtained with algebraic models [11] because it follows the same domain analysis.

In FDL*, the main idea is to represent features of a given system, so the most important and basic element of the domain is the feature, which has its name as a property. It is a basic solution and we define it by using the constructor atomic : String $\rightarrow$ Feature. There is also a special element in the domain that represents a null feature, represented as null : Feature.

The ways to combine the features and generate new ones are given below as function signatures.

opt : Feature $\rightarrow$ Feature
an optional feature
all : Feature $*$ Feature $\rightarrow$ Feature
two mandatory features
one-of : Feature $*$ Feature $\rightarrow$ Feature
two alternative features
more-of : Feature $*$ Feature $\rightarrow$ Feature
a non-exclusive selection
feature_name : Feature
the name of a composite feature that appears elsewhere and is replaced by its definition.

We describe an implementation of FDL* into the pure object-oriented programming language Smalltalk. We decided to use the *compiler generator* implementation approach [13] by using a tool called SmaCC, which is a Yacc-based compiler generator. The semantics of FDL was easy to define following the design results. Figure 3 shows the representation of features in terms of a class hierarchy. A system description is composed by several composite feature definitions (at least one). We describe the different definitions by simple equations where the left-hand side of each one is the name of the composite feature and the right-hand side is the composite feature itself. The definition name could appear inside other feature definitions, or it is the main feature definition that

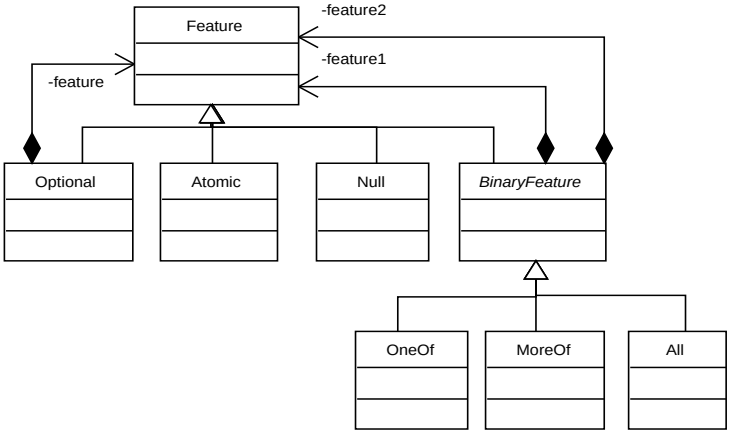

Fig. 3. Representation of features using a class hierarchy

does not appear anywhere. By convention, the main feature description (the one describing the whole system) is placed first in the specification. An example of a system configuration for a typical rock band is given in Figure 4.

In the original definition of FDL there is an implicit transformer that joins all feature definitions into one, returning a regular form of a program. We define it as: regularForm: list of Feature → Feature. This transformer replaces occurrences of named features by their definition, and gives evidence of the compositional semantics of features. This transformer and the ones defined later serve as a guide in the language implementation — they do not form part of the object language.

This transformer is implemented as a method over the class hierarchy of Figure 3 following the *Composite* pattern. The FDL* parser creates an instance of the class *SpecConstructor* with all the definitions and passes the rest of the definitions to the first one. The main feature receives the rest of the definitions and makes regularForm work.

The parser definition is given in Figure 7. The class for representing named features is not necessary because they are automatically replaced by the corresponding definition.

Once the regular form is obtained, it can be obtained the *disjunctive normal form* (DNF) which, when different from null or atomic features, has a structure described by:

$$\text{one-of}\begin{pmatrix} \text{all}(A_{1_1},\ldots,A_{1_n}) \\ \vdots \\ \text{all}(A_{m_n},\ldots,A_{m_n}) \end{pmatrix}$$

The DNF gives the user all possible configurations for the system. To get the DNF, it is necessary to define dnf : Feature → Feature. In Figure 5 it can be seen the definition for dnf transformer, giving equations for the different cases of features. As a consequence of the recursive structure of features, the transformer is also defined recursively.

The equations given in Figure 5 guide the implementation of static semantics of the language. These equations are translated to different dnf method definition in the features hierarchy of Figure 3. As a short example, the dnf method definition for one-of features is:

```
OneOf>>dnf
   self feature1: self feature1 dnf.
   self feature2: self feature2 dnf.
```

Expressions in DNF may have repeated features. For that reason, we define a normalized version of every feature in disjunctive normal form, that allows to remove those unnecessary duplications. A feature expression f is normalized if it has the following form:

$$\text{one-of}\begin{pmatrix} \text{all}(A_{1_1},\ldots,A_{1_n}) \\ \vdots \\ \text{all}(A_{m_n},\ldots,A_{m_n}) \end{pmatrix}$$

where $A_{i_j} \neq A_{i_k}$ for $j,k \in \{1,\ldots,i_n\}$ and $i \neq j$ and $\text{all}(A_{i_1},\ldots,A_{i_n}) \neq \text{all}(A_{j_1},\ldots,A_{j_n})$ for $i,j \in \{1,\ldots,m\}$ and $i \neq j$

In this case, the equations that define normalize, the transformer of features in disjunctive normal form, are shown in Figure 6.

In the Smalltalk implementation, this transformer is defined in the classes *All*, *Atomic*, *Null* and *OneOf*. Bellow is an example of code for the case of all features:

```
All>>normalize
   | temp res y |
   temp:= self feature1 allsToList.
   y := self feature2 allsToList.
   res := self remDupsInList: temp and:y.
   self become: (self createAllTreeFrom: res).

All>>allsToList
   | res |
   res := OrderedCollection withAll:
                        self feature1 allsToList.
   res addAll: self feature2 allsToList.
   ^ res
```

```
rockband    =  all vocals (all strings (all percussion opt(atomic keyboard)))
vocals      =  all (atomic singer) (opt (atomic chorus))
strings     =  more-of (atomic guitar) (atomic bass)
percussion  =  all (atomic drums) (opt (atomic tambourine))
```

Fig. 4. Specification of a basic rock band configuration

```
dnf null                 =  null
dnf (atomic f)           =  atomic f
dnf (opt f)              =  one-of null f
dnf (one-of f1 f2)       =  one-of (dnf f1) (dnf f2)
dnf (more-of f1 f2)      =  one-of (dnf f1) (one-of (dnf (all (dnf f1) (dnf f2))) (dnf f2))
dnf (all f1 f2)          =  distribute (all (dnf f1) (dnf f2))
distribute (all (one-of f11 f12) f2)  =  one-of (distribute (all f11 f2)) (distribute (all f12 f2))
distribute (all f1 (one-of f21 f22))  =  one-of (distribute (all f1 f21)) (distribute (all f1 f22))
distribute f                          =  f -- f being different from an all feature
```

Fig. 5. Transformer definition for obtaining the disjunctive normal form

```
normalize Null              =  Null
normalize (Atomic f)        =  Atomic f
normalize (All f1 f2)       =  allsFromList (remDupsInList
                                    (allsToList f1 ++ allsToList f2))
normalize (OneOf f1 f2)     =  oneOfsFromList (remDupsInList
                                    (oneOfsToList normalize f1 ++
                                     oneOfsToList normalize f2))
allsToList Null             =  {}
allsToList (Atomic f)       =  {Atomic f}
allsToList (All f1 f2)      =  (allsToList f1) ++ (allsToList f2)

oneOfsToList Null           =  {Null}
oneOfsToList (Atomic f)     =  {Atomic f}
oneOfsToList (All f1 f2)    =  {All f1 f2}
oneOfsToList (OneOf f1 f2)  =  (oneOfsToList f1) ++ (oneOfsToList f2)

allsFromList xs             =  treeFromList All xs

oneOfsFromList xs           =  treeFromList OneOf xs

treeFromList n {}           =  Null
treeFromList n {f}          =  f
treeFromList n (f1:fs)      =  n f1 (treeFromList n fs)

remDupsInList {}            =  {}
remDupsInList (x:xs)        =  if x ‘elem‘ xs then remDupsInList xs
                               else x: remDupsInList xs
```

Fig. 6. Transformer equations for obtaining the normalized form

```
All>>remDupsInList: aFeatureList
              and: otherfeatureList
   | res |
   res :=OrderedCollection new.
   aFeatureList do:[:f | (res includes: f)
                      ifFalse:[res addLast:f]].
   otherfeatureList do:[:f | (res includes: f)
                      ifFalse:[res addLast:f]].
   ^ res

Atomic>>allsToList
   ^ OrderedCollection with: self.

Null>>allsToList
   ^ OrderedCollection new.
```

Finally, we can restrict all possible configuration by using constraints. We do not give sample codes from this feature, but we define it just to define completely the language.

A constraint could be one of the following, defined by combinators:

$$\text{requires} : \text{AtomicFeature} * \text{AtomicFeature} \rightarrow \text{Constraint}$$

a diagram constraint establishing that if the first

feature is present, then the second one should also be present.

excludes : AtomicFeature $*$ AtomicFeature $\rightarrow$ Constraint

a diagram constraint establishing that if the first feature is present, then the second one should not be present.

include : AtomicFeature $\rightarrow$ Constraint

a user constraint establishing that the feature should be present.

exclude : AtomicFeature $\rightarrow$ Constraint

a user constraint establishing that the feature should not be present.

Note that AtomicFeatures can be represented as simple strings.

The following constraint definitions can be added to the feature specification of Figure 4: bass requires drums, include guitar, include drums. Observe that infix notation is used, for clarity.

Part of the final SmaCC specification (which includes constraints) can be seen in Figure 7. The designer could make an adjustment of language constructs, to adapt them to a more convenient and familiar writing style for end-users or just to make implementation less costly.

To give an example of language extension by using combinators, we extend the FDL* language, as suggested in the original FDL paper [16], by adding boolean expressions as constraints, associating numeric values with atomic features (e.g., HorsePower = 75), and adding relational operators (e.g., HorsePower $>$ 100).

We define the constructor numberedFeature : Feature $*$ Number $\rightarrow$ Feature. Note that this constructor gives a feature as the result. It implies that these new kinds of features are added in FDL* basic semantics and they can be used in the same way as old feature forms. Of course, the designer must analyze if this modification alters the original semantics. In the Smalltalk implementation this kind of feature is simply a new class in the hierarchy of Figure 3.

For the new forms of constraints, we define the following constructors (that are actually combinators):

$\&\&$: Constraint $*$ Constraint $\rightarrow$ Constraint
$\|$: Constraint $*$ Constraint $\rightarrow$ Constraint
$\neg$: Constraint $\rightarrow$ Constraint
$->$: Constraint $*$ Constraint $\rightarrow$ Constraint
$=$: AtomicFeature $*$ Number $\rightarrow$ Constraint
$>$: AtomicFeature $*$ Number $\rightarrow$ Constraint
$<$: AtomicFeature $*$ Number $\rightarrow$ Constraint

The meaning of the new elements is self evident. As new features are evaluated into booleans, they can be assimilated by the original semantics of features. As new features are evaluated into booleans, they can be assimilated by the original semantics of features. In addition, infix notation can be used for clarity in some of the new combinators.

Known uses

Combinators pattern were widely used in functional programming. Here we present briefly some examples:

Hallgren and Carlsson described a toolkit for the construction of GUIs based on fudgets (functional widgets) and a set of combinators (serial and parallel combinations and loops) [7].

John Hughes described the design of a pretty-printing library using combinators. The author concentrates on two ways of transforming a formal algebraic specification into an implementation, starting with simple examples and working up to the library [9].

Haskore is a collection of Haskell modules designed for expressing musical structures. In Haskore, musical objects consist of primitive notions such as notes, operations to transform musical objects such as transpose and tempo-scaling, and operations to combine musical objects, such as concurrent and sequential composition. [8]

Lava is a tool to assist circuit designers in specifying, designing, verifying and implementing hardware. The circuits and the way they are built are described using combinators [1].

HaXML is a collection of utilities for parsing, filtering, transforming, and generating XML documents using Haskell based on combinators [18].

WASH is a family of EDSLs for server-side web scripting with sessions, compositional forms, and graphics that it is implemented as a combinators library [15].

Many authors have written about parsing combinators in functional programming. The work of Fokker defining a combinators library is such an example, written in Haskell [5]. But functional languages are not the only ones that were used for writing parser combinators. A good example is a *sourceforge* project called Spirit [4]. Spirit is an object-oriented recursive descent parser framework implemented using template meta-programming techniques [3][17]. It consists of a set of basic parsers and parser combinator which enables a target grammar to be written exclusively in C++. Parser objects are composed through operator overloading and the result is a backtracking, top down parser that is capable of parsing rather ambiguous grammars. As an example, a short part of a Pascal language parser definition is:

```
program = programHeading >> block >> DOT;
 programHeading = PROGRAM >> identifier >>
        LPAREN >> fileIdentifier >>
            *( COMMA >> fileIdentifier ) >>
        RPAREN >> SEMI;
 fileIdentifier = identifier;
 block= *( labelDeclarationPart
      | constantDefinitionPart
      | typeDefinitionPart
      | variableDeclarationPart
      | procedureAndFunctionDeclarationPart )
      >> statementPart;
```

Luca Cardelli and Rowan Davies developed a system for web computing in Java using combinators [2]. They defined several combinators and the language was implemented using the object-oriented language Java.

```
%right "all";
%right "one_of" "more_of";
Diagram: Feature_Definitions 'dl' ConstraintsDefinitions 'cs'
          {SpecConstructor new: dl constraints: cs}
ConstraintsDefinitions: {OrderedCollection new}
       | "Constraints:" Feature_constraints 'cs' {cs};
Feature_Definitions: {OrderedCollection new}
       | Feature_Definitions Feature_definition {'1' add: '2'; yourself};
Feature_definition: <identifier> 'name' "=" Feature_expression 'f'
                      {Definition new: name value feature: f};
Feature_expression: "atomic" <identifier> 'name' {Atomic new: name value}
       | <identifier> 'name' {name value}
       | "opt" "(" Feature_expression 'f' ")" {Optional new: f}
       | "one-of" Feature_expression 'f1' Feature_expression 'f2'
                     {OneOf new:f1 and:f2}
       ...
 Feature_constraints: {OrderedCollection new}
        | Feature_constraints Feature_constraint {'1' add: '2';yourself}
        | Feature_constraints Feature_constraint "," {'1' add: '3';yourself};
Feature_constraint: <identifier> 's1' "requires" <identifier> 's2'
                        {Requires new: s1 value with: s2 value}
       | "include" <identifier> 's1' {Include new: s1 value}
       ...
```

Fig. 7. Final SmaCC specification for FDL language parser

Related patterns

This pattern is related with the pattern for functional programming implementation [14] and with the *Composite* pattern [6] from object-oriented programming in the way they describe elements from the domain that have compositional semantics.

ACKNOWLEDGMENTS

We thank Marjan Mernik for his careful reading and comments in an advanced version of the paper.

REFERENCES

[1] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in Haskell. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 174–184. ACM Press, 1998.

[2] Luca Cardelli and Rowan Davies. Service combinators for web computing. *Software Engineering*, 25(3):309–316, 1999.

[3] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.

[4] Joel de Guzman. Spirit — parser combinators for C++. http://spirit.sourceforge.net/.

[5] J. Fokker. Functional parsers. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming: 1st International Spring School on Advanced Functional Programming Techniques*, pages 1–23. Springer, Berlin, 1995.

[6] E. Gamma, R. Helm, and R. Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.

[7] T. Hallgren and M. Carlsson. Programming with Fudgets. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming: 1st International Spring School on Advanced Functional Programming Techniques*, pages 137–182. Springer, Berlin, 1995.

[8] Paul Hudak. Haskore music tutorial. In *Advanced Functional Programming, Second International School-Tutorial Text*, pages 38–67. Springer-Verlag, 1996.

[9] John Hughes. The design of a pretty-printing library. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 53–96. Springer-Verlag, 1995.

[10] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Eng. Institute, Carnegie Mellon University, November 1990.

[11] Pablo E. Martínez López and Jerónimo Irazábal. An algebra for describing features. In *Proceedings of the XXXI Conferencia Latinamericana de Informtica (CLEI'05)*, October 2005.

[12] Brian McNamara and Yannis Smaragdakis. Functional Programming in C++ using the FC++ library. *SIGPLAN Notices*, 36(4):25–30, 2001.

[13] M. Mernik, J. Heering, and A. Sloane. When and how to develop domain-specific languages. Technical report, University of Maribor, CWI Amsterdam, and Macquarie University, 2003.

[14] P. Mocciola and Pablo E. Martínez López. Design patterns for functional programming. In *3rd Latin-American Conf. on Functional Programming*, 1999.

[15] Peter Thiemann. WASH/CGI: Server-Side Web Scripting with Sessions and Typed, compositional forms. In *PADL*, pages 192–208, 2002.

[16] Arie van Deursen and Paul Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.

[17] T. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995.

[18] Wallace, M., Runciman, C.: Haskell and XML: Generic combinators or type-based translation? In: Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99). Volume 34–9., N.Y., ACM Press (1999) 148–159

[19] Jones, S. P., (editors), J. H.: Haskell 98: A non-strict, purely functional language. http://www.haskell.org/onlinereport/ (February 1999)