# The design and implementation of Object Grammars

Tijs Van Der Storm, William R. Cook, Alex Loh



# The Design and Implementation of Object Grammars

Tijs van der Storm[a,*], William R. Cook[b], Alex Loh[b]

[a] *Centrum Wiskunde & Informatica (CWI), Science Park 123, 1098 XG Amsterdam, The Netherlands*
[b] *University of Texas at Austin, 1 University Station, Austin, Texas 78712, US*

**Abstract**

An *Object Grammar* is a variation on traditional BNF grammars, where the notation is extended to support declarative bidirectional mappings between text and object graphs. The two directions for interpreting Object Grammars are *parsing* and *formatting*. Parsing transforms text into an object graph by recognizing syntactic features and creating the corresponding object structure. In the reverse direction, *formatting* recognizes object graph features and generates an appropriate textual presentation. The key to Object Grammars is the expressive power of the mapping, which decouples the syntactic structure from the graph structure. To handle graphs, Object Grammars support declarative annotations for resolving textual names that refer to arbitrary objects in the graph structure. Predicates on the semantic structure provide additional control over the mapping. Furthermore, Object Grammars are compositional so that languages may be defined in a modular fashion. We have implemented our approach to Object Grammars as one of the foundations of the Ensō system and illustrate the utility of our approach by showing how it enables definition and composition of domain-specific languages (DSLs).

*Keywords:* domain-specific languages; model-driven development; language composition; syntax definition

## 1. Introduction

A grammar is traditionally understood as specifying a language, defined as a set of strings. Given such a grammar, it is possible to recognize whether a given string is in the language of the grammar. In practice it is more useful to actually parse a string to derive its meaning. Traditionally parsing has been defined as an extension of the more basic recognizer: when parts of the grammar are recognized, an action is invoked to create the (abstract) syntax tree. The actions are traditionally implemented in a general-purpose programming language.

In this paper we introduce *Object Grammars*: grammars that specify mappings between syntactic presentations and graph-based object structures. *Parsing* recognizes syntactic features and creates object structures. Object grammars include declarative directives indicating how to create cross-links between objects, so that the result of parsing can be a graph. *Formatting* recognizes

*Corresponding author
*Email addresses:* storm@cwi.nl (Tijs van der Storm), wcook@cs.utexas.edu (William R. Cook), alexloh@cs.utexas.edu (Alex Loh)

object graph features and creates a textual presentation. Since formatting is not uniquely specified, an Object Grammar can include formatting hints to guide the rendering to text.

The second problem addressed in this paper is modularity and composition of Object Grammars. Our goal is to facilitate construction of domain-specific languages (DSLs). It is frequently desirable to reuse language fragments when creating new languages. For example, a state machine language may require an expression sub-language to represent constraints, conditions, or actions. In many cases the sub-languages may also be extended during reuse. We present a generic merge operator that covers both reuse and extension of languages.

The contributions of this paper can be summarized as follows:

- We introduce the *Object Grammar* formalism to describe mappings from textual notation to object graphs.
- Cross references in the object structure are resolved using *declarative paths* in the Object Grammar.
- Complex mappings can be further controlled using *predicates*.
- We show that Object Grammars are both *compositional* and *bidirectional*.
- We present an interpretative implementation of the general parsing algorithm GLL [56].
- The entire system is self-describing and bootstrapped within itself.

The form of Object Grammars presented in this paper is one of the foundations of Ensō, a new programming system for the definition, composition and interpretation of external DSLs. At the time of writing, Ensō is implemented in the Ruby programming language [21]. For more information and links to the source code, the reader is referred to `http://www.enso-lang.org`. This paper extends and revises [59] with an extended introduction motivating the design of Object Grammars, additional details on the implementation of Object Grammars in Ensō (Section 5), an additional case-study to evaluate Object Grammars (Section 6), and additional directions for further research (Section 8).

### *1.1. Ensō: Application Software = Models + Interpreters*

Ensō is a programming system for the definition and interpretation of executable specification languages or *models*. Examples of such languages include languages for describing data models (schemas), GUIs, security policy, Web applications and syntax (grammars). The goal of Ensō is to develop application software by combining such languages. The runtime behavior of an application is defined by composing interpreters for these languages. For instance, an interpreter for a GUI language renders a specification of a GUI on the screen and ensures that user events are interpreted in the desired way.

All data in the Ensō system is described by a schema, including schemas themselves. A schema is a class-based information model, similar to UML Class Diagrams [46], Entity-Relationship Diagrams [12] or other meta-modeling formalisms (e.g., [7, 28, 32]). Schemas are interpreted by a data manager. This leads to the perspective of *managed data*: the way models are created, queried, or modified is managed by an interpreter of schemas (called a "factory") [41]. The factory is used to create objects of types declared in the schema, to update fields of such objects, and to raise an error if undeclared fields are accessed. An example of interpreter composition is adding a security aspect to the factory. The interpreter of security policies acts as a proxy for

the factory, only passing requests through to the normal factory if the current user is allowed to read or write a particular property. A similar example is discussed in Section 5 where a schema interpreter is extended to implement maximal sharing [27].

Object Grammars are Ensō's language for describing the textual appearance of models, including the textual appearance of object grammars themselves. Ensō is self-hosted so that all aspects of the system (models, interpreters) are open for modification and extension. It is not a goal to provide a definitive set of DSLs for building application software. Rather, Ensō is a platform for the creation, adaptation and extension of DSLs, including the foundational languages, like schema and grammar. It is therefore important to realize that the version of Object Grammars in this paper is not aimed at parsing all (existing) languages. The capabilities of Object Grammars as described here represent the current state in the evolution of the Ensō system. Although the concepts to be developed later in this paper are general, the current implementation makes trade-offs that reflect our current needs for defining languages in Ensō. The self-describing aspect of Ensō allows the way the structure of models is described using schemas to be modified if needed, and the same is true of Object Grammars. The interpreters used to execute these languages are open for extension as well, – they are part of the package. In Section 5 we will see how extensions of the core models and interpreters are used to implement parsing.

### *1.2. Grammars and Models*

In textual modeling [44] models are represented as text, which is easy to create, edit, compare and share. To unlock their semantics, textual models must be parsed into a structure suitable for further processing, such as analysis, (abstract) interpretation or code generation. Many domain-specific models are naturally graph structured. Well-known examples include state machines, work-flow models, petri nets, network topologies, class diagrams and grammars. Nevertheless, traditional approaches to parsing text have focused on tree structures. Context-free grammars, for instance, are conceptually related to algebraic data types. As such, existing work on parsing is naturally predisposed towards expression languages, not modeling languages. To recover a semantic graph structure, textual references have to be resolved in a separate name-analysis phase.

Object Grammars invert this convention, taking the semantic graph structure (the model) rather than the parse tree as the primary artifact. Hence, when a textual model is parsed using an Object Grammar, the result is a graph. Where the traditional tree structure of a context-free grammar can be described by an algebraic data type, the graphs produced by Object Grammars are described by a schema.

There is, however, an *impedance mismatch* between grammars and object-oriented schemas. Grammars describe both syntactic appearance and syntactic tree structure. Schemas, on the other hand, describe semantic graph structures. As a result, any attempt at bridging grammars and models, has to make certain trade-offs to overcome the essential difference between grammars and models. Previous work has suggested the use of one-to-one mappings between context-free grammar productions and schema classes [1, 71]. However, this leads to tight coupling and synchronization of the two formats. A change to the grammar requires a change to the schema and vice versa. As a result, both grammar and schema have to be written in such a way that this correspondence is satisfied.

Another trade-off is concerned with how graph structures are derived from the essentially tree-based grammar. In traditional style parser generators, such as Yacc [31] or ANTLR [49], the semantic structure is created by writing general purpose code. The flexibility of such semantic actions has the advantage that any desired structure can be created, including graph-like models.

| Requirement | Design decision(s) |
|---|---|
| Unified formalism | Integrated data binding |
| Graph-based models | Constructs for reference resolving |
| Flexible | Asynchronous data binding, predicates, and formatting hints |
| Bidirectional | Declarative data binding |
| Compositional | Generalized parsing |
| Extensible | Self-described and bootstrapped |

Table 1: Relating requirements and design decisions

However, general purpose code is generally not invertable and as a result constructed trees or graphs cannot be automatically transformed back to text. The language workbench Xtext provides a generic lookup mechanism to resolve references based on globally unique identifiers [20]. This basic lookup mechanism can be used to automatically derive formatters. However, it also restricts the name resolution rules of a language implemented in Xtext. The lookup rules can be customized by providing name resolution code in Java. However, the semantics of references is then external to the grammar specification itself and bidirectionality is compromised. Object Grammars, on the other hand, allow more flexibility than one-to-one mappings, richer name lookup semantics than, e.g., Xtext, while still preserving bidirectionality. As such they represent a unique point in the design space, – inspired by earlier work, but motivated by a unique combination of requirements.

### *1.3. Requirements and Design Decisions*

Ensō is an extensible platform for the definition and composition of DSLs. Object Grammars serve to define the "textual user interface" of such languages. Below we elaborate on the requirements that have guided the design of Object Grammars and discuss the decisions and trade-offs that have shaped the design of Object Grammars. How the individual features of the approach are related to earlier work is analyzed in more depth in Section 7. A summary of how the requirements are addressed is shown in Table 1.

*Unified formalism.* A single, unified formalism should be sufficient to define the textual syntax of a language the structure of which is defined in a schema. This requirement implies that how the object structure is created after parsing is specified within the grammar itself. The Object Grammar formalism features constructs to specify object construction and field binding directly in the grammar. Furthermore, Object Grammars do not require a separate scanning phase. Instead of allowing the specification of lexical syntax explicitly, we have opted for a fixed set of token types, which correspond to common lexemes in programming languages (e.g., numeric literals, string literals etc.). Furthermore, the tokens correspond to the primitive types currently supported by the schema language.

*Graph-based models.* As noted above, the semantic structure of DSLs is often naturally graph-structured. The formalism should allow expressing a textual syntax that concisely and conveniently maps to such structures. To facilitate the construction of graph-like models, fields can be bound to objects at arbitrary locations in the resulting object graph. Such binding is specified using declarative path expressions which locate the target object based on textual names in the

input stream. DSLs are often small, declarative specification languages (context-free grammars are the text book example). Even though reference resolution is an important aspect of most languages, many DSLs do not feature the complex scoping rules of, for instance, Java or C#. Hence, we consider full name analysis involving scopes, name spaces, and imports outside the scope of Object Grammars.

*Flexible.* Because of the impedance mismatch between grammars and schemas, the mapping between them should be flexible and customizable. To achieve this, the construction and field binding constructs may be "asynchronously" interleaved between the syntactic constructs of the formalism, independent of the syntactic structure of the grammar. For instance, there is no implicit relation between non-terminals and rule alternatives on the one hand, and concepts of the schema (types, fields, etc.) on the other hand. Additional semantic predicates can be used to further customize the mapping. Asynchronous data binding and predicates promote separation of concerns and loose coupling: the structure of the grammar can be optimized for readable syntax, whereas the structure of the schema can be optimized for conceptual integrity. Finally, to further customize the output of formatting an object graph to text, Object Grammars can be decorated with formatting directives. This set of directives is not aimed at providing complete control over the output, as in general pretty printing frameworks, but are sufficient to produce readable, indented renderings.

*Bidirectional.* The textual interface is but one of many possible user interfaces to manipulate models. For instance, some models are also conveniently edited using GUI forms or diagram editors. To still be able to store models in a textual format, the grammar formalism should be bidirectional. Bidirectionality is supported if the mapping between syntax and schema is specified using constructs that admit a bidirectional interpretation. The aforementioned constructs – object construction, field binding, paths, and predicates – can be interpreted "backwards" to support formatting. During parsing, object construction, field binding and predicates are *actions* that manipulate the object graph being created, but during formatting they are interpreted as *guards* to guide the formatter through the grammar. Path expressions *locate* the referenced object during parsing, but during formatting, they are *solved* to find the textual name that has to be output.

*Compositional.* To facilitate reuse and extension of languages, the formalism should allow the composition of different languages. Compositionality is informally defined as the ability to combine two Object Grammars in order to process textual models that are specified using the combined syntax. To satisfy this requirement, Object Grammars are built on a foundation of general parsing. As a result, Object Grammars support arbitrary context-free syntax, which is closed under union. Furthermore, general parsing provides a high level of flexibility and expressiveness: the grammar writer does not have to restructure the grammar, for instance to satisfy lookahead restrictions or to avoid left-recursion. General parsing, however, might incur ambiguities. This problem is resolved in a pragmatic way: ambiguous sentences are treated as parse-time errors. In other words, we have traded the static guarantees provided by conventional LR or LL parsing for increased flexibility.

*Extensible.* The grammar formalism itself should be open for extension and modification. In the spirit of Ensō being a platform rather than a tool set we would like to be able to modify, extend, or reuse the Object Grammar formalisms in the same way ordinary DSLs could be extended. One way of achieving this is aiming for a system that is as self-hosting as possible. In fact,

as we will see in Section 4, the formalism of Object Grammars is defined in itself. As such, the current incarnation of Object Grammars in Ensō can be a stepping stone for more advanced Object Grammar formalisms.

### *1.4. Organization*

This paper is organized as follows. Section 2 introduces Object Grammars from the perspective of how they are used to define the syntax of languages. This provides an overview of the features for mapping text to object graphs, including object construction, field binding, path-based references and predicates. Section 3 describes common language composition scenarios and how they are addressed in Ensō by merging Object Grammars. We identify three use cases for composition, which are illustrated using Object Grammar examples. Section 4 elaborates on how object Grammars (and schemas) are described within their own formalism. Section 5 presents the Object Grammar implementation details. We first elaborate upon the notion of interpetation for executing models. Examples of such model interpreters are parsing, formatting, object graph building, and merging, which are presented in detail. The section is concluded by detailing the bootstrap process of Ensō. In Section 6 we evaluate the formalism based on how it is used and reused throughout Ensō. An external case-study in domain-specific modeling serves as a separate evaluation benchmark. Related work is surveyed in Section 7. We position Object Grammars in the area of bridging modelware and grammarware and provide pointers to related work on language composition and self-describing systems. Finally, we conclude the paper in Section 8.

## 2. Object Grammars

An Object Grammar specifies a mapping between syntax and object graphs. The syntactic structure is specified using a form of Extended Backus-Naur Form (EBNF) [72], which integrates regular alternative, sequential, iteration and optional symbols into BNF. Object Grammars extend EBNF with constructs to declaratively construct objects, bind values to fields, create cross links and evaluate predicates.

### *2.1. Construction and Field Binding*

The most fundamental feature of Object Grammars is the ability to declaratively construct objects and assign to their fields values taken from the input stream. The following example defines a production rule named `P` that captures the standard notation `(x, y)` for cartesian points and creates a corresponding `Point` object.

```
P ::= [Point] "(" x:int "," y:int ")"
```

The production rule begins with a *constructor* `[Point]` which indicates that the rule creates a `Point` object. The literals `"("`, `","` and `")"` match the literal text in the input. The *field binding* expressions `x:int` and `y:int` assign the fields `x` and `y` of the new point to integers extracted from the input stream. The classes and fields used in a grammar must be defined in a *schema* [41]. For example, the schema for points is:

```
class Point   x: int  y: int
```

Any pattern in a grammar can be refactored to introduce new non-terminals without any effect on the result of parsing. For example, the above grammar can be rewritten equivalently as

```
P  ::= [Point] "(" XY ")"
XY ::= x:int "," y:int
```

The `XY` production can be reused to set the `x` and `y` fields of any kind of object, not just points.

Operationally, the object graph is obtained in two steps. First the input is parsed using the Object Grammar; if successful, this results in a single, non-ambiguous parse tree annotated with object construction and field binding directives. This phase is equivalent to traditional, context-free parsing. In the second phase, the resulting parse tree is traversed to build the object graph. In the example above this will be a single `Point` object. The details of this process are described in Section 5.3.

The Object Grammars given above can also be used to format points into textual form. The constructor acts as a guard that specifies that only points should be rendered using this rule. The literal symbols are copied directly to the output. The field assignments are treated as selections that format the `x` and `y` fields of the point as integers.

### *2.2. Alternatives and Object-Valued Fields*

Each alternative in a production can construct an appropriate object. The following example constructs either a constant, or one of two different kinds of `Binary` objects. The last alternative does not construct an object, but instead returns the value created by the nested `Exp`.

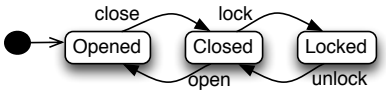
```
Exp ::= [Binary] lhs:Exp op:"+" rhs:Exp
     |  [Binary] lhs:Exp op:"*" rhs:Exp
     |  [Const] value:int
     |  "(" Exp ")"
```

This grammar is not very useful, because it is ambiguous. Although the Ensō parser can handle ambiguous grammars, interpreting an ambiguous parse as an object graph is problematic since it is unclear which derivation is the intended one. To resolve the ambiguity, we use the standard technique for encoding precedence and associativity using additional non-terminals.

```
Term ::= [Binary] lhs:Term op:"+" rhs:Fact   |  Fact
Fact ::= [Binary] lhs:Fact op:"*" rhs:Prim   |  Prim
Prim ::= [Const] value:int                   |  "(" Term ")"
```

This grammar refactoring is independent of the schema for expressions; the additional non-terminals (`Term`, `Fact`, `Prim`) do not have corresponding classes. Ambiguous grammars are not disallowed: as long as individual input strings are not ambiguous there will be no error. The original version thus *can* be used to meaningfully parse fully parenthesized expressions, but the result will be ambiguous otherwise. The second version, however, handles standard expression notation.

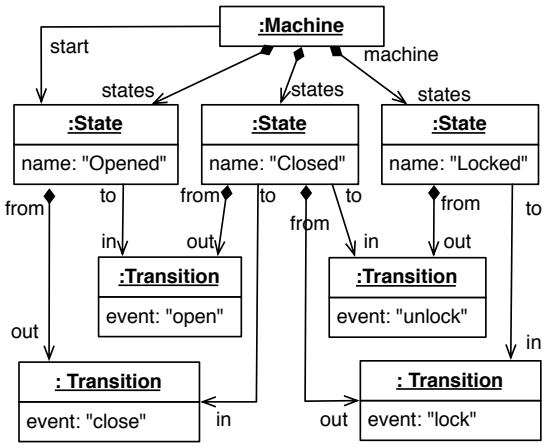
During formatting, the alternatives are searched in order until a matching case is found. For example, to format `Binary(Binary(3,"+",5),"*",7)` as a `Term`, the top-level structure is a binary object with a $*$ operator. The `Term` case does not apply, because the operator does not match, so it formats the second alternative, `Fact`. The first alternative of `Fact` matches, and the left hand side `Binary(3,"+",5)` must be formatted as a `Fact`. The first case for `Fact` does not match, so it is formatted as a `Prim`. The first case for `Prim` also does not match, so parentheses are added and the expression is formatted as a `Term`. The net effect is that the necessary parentheses are added automatically, to format as `(3+5)*7`.




(a)

```
start Opened
state Opened
  on close go Closed
state Closed
  on open go Opened
  on lock go Locked
state Locked
  on unlock go Closed
```

(b)




(c)

Figure 1: (a) Example state machine in graphical notation, (b) the state machine in textual notation, and (c) the internal representation of the state machine in object diagram notation

### *2.3. Collections*

Object Grammars support regular symbols to automatically map collections of values. For example, consider this grammar for function calls:

```
C ::= [Call] fun:id "(" args:Exp* @"," ")"
```

The regular repetition grammar operator $*$ may be optionally followed by a separator using `@`, which in this case is a comma. The `args` field of the `Call` class is assigned objects created by zero-or-more occurrences of `Exp`. A collection field can also be explicitly bound multiple times, rather than using the $*$ operator. For example, `args:Exp*` could be replaced by `Args?` where `Args ::= args:Exp (","Args)?`.

### *2.4. Reference Resolving*

In order to explain path-based reference resolution in Object Grammars, it is instructive to introduce a slightly more elaborate example. Consider a small DSL for modeling state machines. Figure 1 displays three representations of a simple state machine representing a door that can be opened, closed, and locked. Figure 1(a) shows the state machine in graphical notation. The same state machine is rendered textually in Figure 1(b). Internally, the machine itself, its states and the transitions are all represented explicitly as objects. This is illustrated in the object diagram given in Figure 1(c).

The object diagram conforms to the State Machine schema given in Figure 2. The schema consists of a list of named classes, each having a list of fields defined by a name, a type, and some optional modifiers. For example, the `Machine` class has a field named `states` which is a set of `State` objects. The $*$ after the type name is a modifier that marks the field as many-valued. The # annotation marks a field as a primary key, as is the case for the `name` field of the `State` class.

```
class Machine
  start  : State
  states ! State*
```

```
class State
  machine: Machine / states
  name   # str
  out    ! Transition*
  in     : Transition*
```

```
class Transition
  event # str
  from  : State / out
  to    : State / in
```

Figure 2: Schema defining the structure of state machine object graphs

```
start M
M ::= [Machine] "start" \start:<root.states[it]> states:S*
S ::= [State] "state" name:sym out:T*
T ::= [Transition] "on" event:sym "go" to:<root.states[it]>
```

Figure 3: Object Grammar to parse state machines

As a result, state names must be unique and the `states` field of `Machine` can be indexed by name. The `/` annotation after the `machine` field indicates that the `machine` and `states` are *inverses*, as are `from`/`out` and `to`/`in`. The `!` modifier indicates that the field is part of the *spine* (a minimal spanning tree) of the object graph. All nodes in a model are assumed to be uniquely reachable by following just the spine fields. The spine allows visiting each object in the object graph in a predictable way. This is useful for generic mapping operations on models, such as printing and serialization. Without the distinction between spine fields and non-spine fields, encoded references could end up in the output at arbitrary locations based on the specific traversal strategy of the operation (e.g., depth-first vs breadth-first). Currently such operations simplify traverse the spine and treat all other object fields as cross-links. Not that the `states` field is on the spine, whereas the `start` field is not. The object pointed to by `start`, however, is required to be included in the set of all `states`.

The schema could potentially be derived from the grammar, but we prefer to specify them both separately: schemas may have additional structure in the form of class inheritance, and computed fields, or other meta-data, which may be unrelated to the Object Grammar.

The textual representation in Figure 1(b) uses *names* to represent links between states, while the graphical presentation in Figure 1(a) uses graphical edges so names are not needed. When humans read the textual presentation in Figure 1(b), they immediately resolve the names in each transition to create a mental picture similar Figure 1(a).

Figure 3 shows an Object Grammar for state machines[1]. It uses the reference `<root.states[it]>` to look up the start state of a machine and to find the the target state of a transition. The path `root.states[it]` starts at the root of the resulting object model, as indicated by the special variable `root`. In this case the root is a `Machine` object, since `M` is the start symbol of the grammar, and the `M` production creates a `Machine`. The path then navigates into the field `states` of the machine (see Figure 2), and uses the identifier from the input stream (`it`) to index into the keyed collection of all states. The same path is used to resolve the `to` field of a transition to the target state.

In general, a reference `<`*p*`>` represents a lookup of an object using the path *p*. Parsing a

[1]The field label `start` is escaped using \ because `start` is a keyword in the grammar of grammars; cf. Section 2.7.

```
start Schema
Schema     ::= [Schema] types:TypeDef* @/2
TypeDef    ::= Primitive | Class
Primitive  ::= [Primitive] "primitive" name:sym
Class      ::= [Class] "class" name:sym ClassAnnot /> defined_fields:Field* @/ </
ClassAnnot ::= Parent?
Parent     ::= "<" supers:Super+ @","
Super      ::= <root.classes[it]>
Field      ::= [Field] name:sym.Kind type:<root.types[it]> Mult? Annot?
Kind       ::= "#"  {key == true}
             | "##" {key == true and auto == true}
             | "!"  {traversal == true}
             | ":"
Mult       ::= ."*" {many == true and optional == true}
             | ."?" {optional == true}
             | ."+" {many == true}
Annot      ::= "/" inverse:<this.type.fields[it]>
             | "=" computed:Expr
```

Figure 4: Schema Grammar

reference always consumes a single identifier, which can be used as a key for indexing into keyed collections. Binding a field to a reference thus results in a cross-link from the current object to the referenced object.

The syntax of paths is reused from a general grammar for expressions, which includes syntax for field dereferencing, and indexing into collections. A path is anchored at the current object (`this`), its parent according to the spine of the graph (`parent`), or at the root (`root`). In the context of an object a path can descend into a field by post-fixing a path with `.` and the name of the field. If the field is a collection, a specific element can be referenced by indexing in square brackets. The special variable `it` represents the string-typed value of the identifier in the input stream that represents the reference name.

The grammar of schemas, given in Figure 4, illustrates a more complex use of references. To lookup inverse fields, it is necessary to look for the field within the class that is the type of the field. For example, in the state machine schema in Figure 1(b), the field `from` in `Transition` has type `State` and its inverse is the `out` field of `State`. The path for the type is `type:<root.types[it]>`, while the path for the inverse is `inverse:<this.type.fields[it]>`, which refers to the type object.

To format a path, for example `root.states[it]` in Figure 3, the system solves the equation `root.states[it]`=*o* to compute `it` given the known value *o* for the field. The resulting name is then output, creating a symbolic reference to a specific object.

### *2.5. Predicates*

The mapping between text and object graph can further be controlled using predicates. Predicates are constraint expressions on fields of objects in the object graph. During parsing, the

```
class Schema
  types ! Type*

class Class < Type
  supers         : Class*
  subclasses     : Class* / supers
  defined_fields ! Field*
  fields         : Field*
    = all_fields.select()
         {|f| !f.computed}
  all_fields     : Field*
    = supers.flat_map()
         {|s| s.all_fields }
            .union(defined_fields)
```

```
class Type
  name   # str
  schema : Schema / types

class Primitive < Type

class Field
  name     # str
  owner    : Class / defined_fields
  type     : Type
  inverse  : Field? / inverse
  computed ! Expr?
  optional : bool
  many     : bool
  key      : bool
  traversal: bool
```

Figure 5: Schema Schema

values of these fields are updated to ensure these constraints evaluate to `true`. Conversely, during formatting, the constraints are interpreted as conditions to guide the search for the right rule alternative to format an object.

Predicates are useful for performing field assignments that are difficult to express using basic bindings. For instance, Ensō grammars have no built-in token type for boolean values to bind to. To write a grammar for booleans, one can use predicates as follows:

```
Bool ::= [Bool] "true"  { value==true }
       |  [Bool] "false" { value==false }
```

Predicates are enclosed in curly braces. When the parser encounters the literal "`true`" it creates a `Bool` object and sets its `value` field to true. Alternatively, when encountering the literal "`false`" the value field is assigned false.

When formatting a `Bool` object, the predicates act as guards. The grammar is searched for a constructor with a fulfilled predicate or no predicate at all. Thus, a `Bool` object with field `value` set to true prints "`true`" and one with field `value` set to false prints "`false`".

A more complex example is shown in the Schema Grammar of Figure 4. The classes and fields used in the grammar are defined in the Schema Schema shown in Figure 5: it defines the structure of schemas, including the structure of itself. Note that this schema introduces subclassing using <: both `Primitive` and `Class` are subclass of `Type`. Furthermore, the Schema Schema uses the computed fields feature of the schema language to obtain the set of all fields (both defined and inherited) from a certain class. Both the `fields` and `all_fields` are accompanied by a Ruby-style expression computing their value in terms of other fields. The expressions in curly braces are lambda expressions passed to the collection methods `select` and and `flatmap`. Note in Figure 4 how the `inverse` field is bound by querying the computed field `fields`.

The production rule for `Mult` assigns the boolean fields `many` and `optional` in different ways. For instance, when a field is suffixed with the modifier "∗", both the `many` and `optional` fields are assigned to values that make the predicate true; in this case both `optional` and `many` are set to true. Conversely, during formatting, *both* `many` and `optional` must be true in the model in order to select this branch and output "∗".

### *2.6. Formatting Hints*

Object Grammars are bidirectional: they are used for reading text into an object structure and for formatting such structure back to text. Since object structures do not maintain the layout information of the source text, formatting to text is in fact pretty-printing, and not unparsing: the formatter has to invent layout. The default formatting simply inserts a single space between all elements. The layout can be further controlled by including formatting hints directly in the grammar. There are three such hints: suppress space (`.`), force line-break (`/`) and indent/outdent (`>` and `<` respectively). They are ordinary grammar symbols so may occur anywhere in a production.

The following example illustrates the use of `.` and `/`.

```
Exp ::= name:sym | Exp "+" Exp | "(".Exp.")"
Stat ::= Exp.";" | "{" / > Stat* @/ < "}"
```

Spaces are added between all tokens by default, so the dot (`.`) is used to suppress the spaces after open parentheses and before close parentheses around expressions. Similarly, the space is suppressed before the semicolon of an expression-statement. The block statement uses explicit line breaks to put the open and close curly braces, and each statement, onto its own line. Furthermore, each individual statement is indented one level. Note that the `Stat` repetition is *separated by* line-breaks (`@/`) during formatting, but, like all formatting directives, this has no effect on parsing.

### *2.7. Lexical Syntax*

Ensō's Object Grammars have a fixed lexical syntax. This is not essential: Object Grammars can easily be adapted to scannerless or tokenization-based parser frameworks. For Ensō's goal, a fixed lexical syntax is sufficient. Furthermore, it absolves the language designer of having to deal with tokenization and lexical disambiguation.

First of all, the whitespace and comment convention is fixed: spaces, tabs and newlines are interpreted as separators; they do not affect the resulting object graph in any other way. There is one comment convention, `//` to end of line. Second, the way primitive values are parsed is also fixed. In the examples we have seen the **`int`** and **`sym`** symbols to capture integers and identifiers respectively. Additional types are **`real`** and **`str`** for standard floating point syntax and strings enclosed in double quotes.

The symbol to capture alpha-numeric identifiers, **`sym`**, is treated in a special way, since it may cause ambiguities with the keyword literals of a language. The parser avoids such ambiguities in two ways. First, any alpha-numeric literal used in a grammar is automatically treated as a keyword and prohibited from being a **`sym`** token. Second, for both keyword literals and identifiers a longest match strategy is applied. As a consequence, a string `ab` will never be parsed as two consecutive keywords or identifiers, but always either as a single identifier, or as a single keyword. Keywords can be used as identifiers by prefixing them with `\`. An example of this can be seen in the state machine grammar of Figure 3, where the `start` field name is escaped because start is a keyword in grammars.

## 3. Object Grammar Composition

### *3.1. Modular Language Development*

Modular language development presupposes a composition operator to combine two language modules into one. For two grammars, this usually involves taking the union of their production rules, where the alternatives of rules with the same name are combined. To union Object Grammars in such a way, it is also necessary to merge their target schemas so that references to classes and fields in both languages can be resolved. Object Grammar composition facilitates modular language development, language reuse and language extension using a single mechanism of *merging* models. A generic merge operator $\lhd$ combines any pair of models described by the same schema. Since both grammars and schemas are themselves such models, the merge operator can be applied to compose Object Grammars by merging two grammars and merging their respective target schemas. The implementation of $\lhd$ is described in Section 5.5.

The merge operator applied to two Object Grammars $G_1 \lhd G_2$ merges the sets of rules. If a rule is present in both $G_1$ and $G_2$ the production alternatives of $G_2$ are appended to the alternatives of the same rule in $G_1$. When a rule is abstract in either $G_1$ or $G_2$ the result will be a rule with the alternatives of the argument where the rule is not abstract.

Merging two schemas $S_1$ and $S_2$ merges the sets of types in both arguments. Similarly, for a class in both $S_1$ and $S_2$ the fields and super types are merged. The latter can be used, for instance, to let classes in $S_1$ inherit additional fields. The attributes of a field (e.g., type, multiplicity, or key) are taken from the same field in $S_2$.

The merge operator $\lhd$ is powerful enough to facilitate different kinds of language composition. We distinguish the following use cases for language composition:

- *Language reuse*: in this case a particular language is included in another language without modification. An example would be a reusable expression language. The reused language is included as-is, and its semantics is encapsulated.

- *Language extension*: a language including another language, but also adding new constructs to included types, is an instance of language extension. The expression language could be extended by adding new expression variants. The extended language, however, is still eligible for encapsulated language reuse.

- *Language mixin*: if the extended language cannot be used independently we speak of language mixin. The extended language provides open hooks that have to be filled by the extending language. An example is a language mixin for control-flow statements. Such a language would not include primitive/base statements; these have to be provided by the extending language.

All three styles are used throughout the implementation of Ensō. Section 6 discusses these examples in more detail. Below we briefly illustrate how each scenario is addressed by merging.

### *3.2. Language Reuse*

As an example of language reuse, consider the addition of entry conditions to the state machine models described by the schema in Figure2. This change requires reusing a generic expression language, *Expr*. The grammar of Figure 3 is then modified as follows:

```
S ::= [State] "state" name:sym out:T*
   |  [State] "state" name:sym out:T* "when" cond:Expr
abstract Expr
```

A new alternative is added to the `S` rule in order to support state definitions with an entry condition, indicated by the "when" keyword. The abstract rule `Expr` captures the, as of yet absent, expression language. The grammar modification anticipates reuse of the syntax of expressions. Naturally, the state machine schema (Figure 2) is modified accordingly:

```
class State
  machine: Machine / states
  name   # str
  out    ! Trans*
  in     : Trans
  cond   ! Expr?
class Expr
```

The `State` class is extended with a new optional field `cond` which contains the entry condition if present. Since entry conditions are hierarchically contained in states, the field `cond` is marked to be on the spine. The empty class `Expr` is required to bind the reference in the `cond` field.

Let's call this new version of the state machine language $Stm'$. The expression language $Expr$ can now be reused by pairwise composing the grammar and schema of $Stm'$ with the grammar and schema of $Expr$:

$$G_{Stm+Expr} = G_{Stm'} \lhd G_{Expr}$$
$$S_{Stm+Expr} = S_{Stm'} \lhd S_{Expr}$$

When the schemas are merged the empty `Expr` class of $S_{Stm}$ is identified with the `Expr` class of the $S_{Expr}$ and the reference to `Expr` in class `State` is updated accordingly. Similarly, merge on grammar entails that the concrete `Expr` non-terminal from $G_{Expr}$ is added to the result. The $\lhd$ operator ensures that unbound references to `Expr` non-terminal will be updated to point to the new `Expr` non-terminal as well.

### *3.3. Language Extension*

In the previous paragraph we showed how a generic expression language could be reused to add entry conditions to the state machine language by modifying both grammar and schema directly. However, we would like to extend the state machine language without having to modify the existing models. This composition case is handled by the $\lhd$-operator as well.

The manual modifications to the state machine grammar and schema could be encapsulated as separate language modules, $G_{cond}$ and $S_{cond}$, which are defined as follows:

```
S ::= [State] "state" name:sym out:T* "when" cond:Expr
abstract Expr
abstract T
```

```
class State
  cond: Expr?

class Expr
```

Both grammar and schema only define the changed syntax and object structure of states. Note that an abstract rule `T` is now needed to resolve the `T` non-terminal in the `S` production. It is not required to repeat the original fields of class `S` in $S_{Cond}$ since the field sets will unioned by $\lhd$.

The combined language can now be obtained by composing *Stm*, *Cond* and *Expr* languages:

$$G_{Stm+Cond} = G_{Stm} \lhd G_{Cond} \lhd G_{Expr}$$
$$S_{Stm+Cond} = S_{Stm} \lhd S_{Cond} \lhd S_{Expr}$$

### *3.4. Language Mixin*

Language reuse is used to include a language in another language without changing it, whereas language extension allows you to extend the included language. In both cases the included language is stand-alone: its grammar does not include abstract rules and its schema does not have any place-holder classes. Conversely, we speak of language mixin if the included language is not usable without a host. Including a language mixin requires you to bind place-holder elements to concrete definitions.

Consider the following language mixin for C-style conditional expressions, showing both grammar and schema at the same time.

```
Expr ::= [If]  c:Expr "?" t:Expr ":" e:Expr
```

```
class Expr
class If < Expr
! c: Expr
! t: Expr
! e: Expr
```

Although both grammar and schema do not explicitly declare any abstract rules or classes, this language cannot be used in a stand-alone fashion: there are no terminal `Expr` alternatives in the grammar, and no leaf `Expr` classes. This characteristic is typical for language mixins.

However, the state machine language extended with entry conditions can act as a host language to "ground" the mixin:

$$G_{Stm+Cond+If} = G_{Stm} \lhd G_{Cond} \lhd G_{Expr} \lhd G_{If}$$
$$S_{Stm+Cond+If} = S_{Stm} \lhd S_{Cond} \lhd S_{Expr} \lhd S_{If}$$

## 4. Self-Description

### *4.1. Introduction*

The Ensō framework is fully self-describing and Object Grammars are one of the foundations that make this possible. Grammars and schemas are both first-class Ensō models [38], just like other DSLs in the system. In Ensō, all models are an *instance* of a schema, and grammar and schema models are no exception. Schemas are instances of the "schema of schemas", which is in turn an instance of itself (see Figure 5). For grammars the relation is *formatting*. For example, the state machine grammar of Figure 3 *formats* state machine models. Similarly, the grammar of grammars (Figure 6) formats itself. The grammar of schemas (Figure 4) parses and formats schemas. The schema of grammars (Figure 7) instantiates grammars, and is formatted using the grammar of schemas. These four core models and their relations are graphically depicted in Figure 8.

```
start Grammar
Grammar  ::= [Grammar] "start" \start:<root.rules[it]> rules:Rule* @/2
Rule     ::= [Rule] name:sym "::=" arg:Alt
Alt      ::= [Alt] > alts:Create+@(/ "|") <
Create   ::= [Create] "[".name:sym."]" arg:Sequence | Sequence
Sequence ::= [Sequence] elements:Field*
Field    ::= [Field] name:sym.":".arg:Pattern       | Pattern
Pattern  ::= [Value] kind:("int"|"str"|"real"|"sym"|"atom")
           |  [Code] "{" expr:Expr "}"
           |  [Ref] "<".path:Expr.">"
           |  [Lit] value:str
           |  [Call] rule:<root.rules[it]>
           |  [Regular] arg:Pattern."*" Sep? {optional==true and many==true}
           |  [Regular] arg:Pattern."?"      {optional==true}
           |  [Regular] arg:Pattern."+" Sep? {many==true}
           |  [NoSpace] ".".
           |  [Break] "/" (.lines:int | {lines==1})
           |  [Indent] ">" {indent==1}
           |  [Indent] "<" {indent==-1}
           |  "(".Alt.")"
Sep      ::= "@".sep:Pattern
abstract Expr
```

Figure 6: The Grammar Grammar: an Object Grammar that describes Object Grammars

```
class Grammar            start: Rule        rules! Rule*
class Rule               name# str          arg! Pattern
                         grammar: Grammar / rules
class Pattern
class Alt < Pattern      alts! Pattern+
class Sequence < Pattern elements! Pattern*
class Create < Pattern   name: str          arg! Pattern
class Field < Pattern    name: str          arg! Pattern
class Call < Pattern     rule: Rule
class Terminal < Pattern
class Value < Terminal   kind: str
class Ref < Terminal     path! Expr
class Lit < Terminal     value: str
class Code < Terminal    expr! Expr
class NoSpace < Pattern
class Break < Pattern    lines: int
class Indent < Pattern   indent: int
class Regular < Pattern  arg! Pattern       optional: bool
                         many: bool         sep! Pattern ?
class Expr
```

Figure 7: The Grammar Schema

Just like ordinary models, the core models have an in-memory object graph representation, which is then interpreted at runtime. Schemas are interpreted by factories, to create and modify object structures. Similarly, both parsing and formatting are implemented as interpreters of object graphs representing grammars (Section 5). The semantics of the core models is thus defined in the same way as for "ordinary" models.

Self-description provides two additional benefits. First, the interpreters that provide the parsing and formatting behavior for Object Grammars can be reused to parse and format grammars and schemas themselves. The same holds for the factories that interpret a schema to construct object graphs: the schema of schemas is just a schema that allows the creation of schemas, including its own schema. Second, interpreters that are "model generic" can be applied to grammars and schemas as well. One example of such a generic operation is merging, which combines two arbitrary models of the same type. Merging is applied to grammars and schemas to compose language modules. Because the four core models are themselves described by a schema, they are amenable to composition and extension in the same way ordinary grammars and schemas are. This property makes it possible, for instance, that both Schema Schema and Grammar Grammar reuse a generic expression language (cf. Section 3).

The self-describing nature of Ensō poses interesting bootstrapping challenges. How these are addressed in Ensō is described in more detail in Section 5.6.

### *4.2. Grammar Grammar*

The formal syntax of Object Grammars is specified by the Grammar Grammar defined in Figure 6. A grammar consists of the declaration of the start symbol and a collection of production

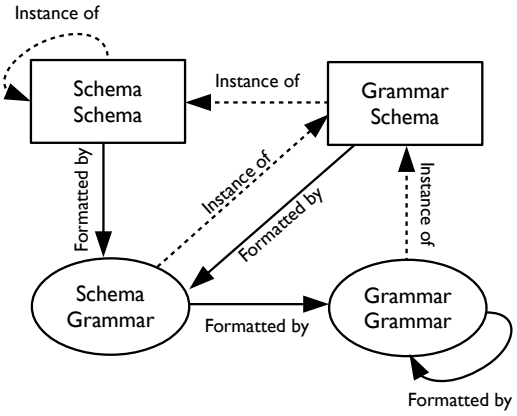



Figure 8: The four core schema and grammar models

rules. A rule is identified by a (non-terminal) `name` and has a body consisting of one or more alternatives separated by (`|`) as defined in the `Alt` rule.

The grammar rules use the standard technique for expressing precedence of grammar patterns, by adding extra non-terminals. An alternative is a `Sequence` of `Patterns` possibly prefixed by a constructor (`Create`), which creates a new object that becomes the current object for the following sequence of patterns. If there is no constructor, the current object is inherited from the calling rule. The `Patterns` in a sequence can be `Field` bindings or syntactical symbols commonly found in grammar notations, such as literals, lexical tokens, non-terminals, regular symbols, and formatting hints.

There is something very elegant and appealing about the concise self-description in the Grammar Grammar. For example the `Create` and `Field` rules both explain and use the creation/binding syntax at the same time. The `Ref` and `Call` rules seem to be inverses of each other, as the body of a `Call` is defined by a reference to a `rule`, and the body of a `Ref` is a call to the non-terminal `Expr`. The normal level and meta-level are also clearly separated, as illustrated by the various uses of unquoted and quoted operators (`|` vs. `"|"`, $*$ vs. `"*"`, etc).

## 5. Implementation

### *5.1. Interpretation*

In contrast to most DSL systems, which are based on code generation, Ensō exclusively uses dynamic interpretation of models. An example is a factory object to interpret a schema to dynamically create objects and assign fields [41]. Parsing and formatting are no exception and are implemented as (a combination of) interpreters of Object Grammars.

These are the three interpreters relevant for the purpose of this paper:

$$parse : Grammar_S \to String \to ParseTree_S \tag{1}$$

$$build : (S : Schema) \to ParseTree_S \to S \tag{2}$$

$$format : Grammar_S \to S \to String \tag{3}$$

```
class Pattern
  prev: Pattern?
  nxt: Pattern?

class Rule < Pattern
  original: Pattern?

class End < Pattern
class EpsilonEnd < End

class Layout < Terminal
```

```
class GSS
  item! Pattern
  pos: int
  edges! Edge*

class Edge
  sppf: SPPF?
  target: GSS
```

```
class SPPF
  starts: int
  ends: int
  type: Pattern

class Node < SPPF
  kids: Pack+

class Leaf < SPPF
  value: str

class Pack
  parent: Node / kids
  type: Pattern
  pivot: int
  left: SPPF?
  right: SPPF
```

Figure 9: Extensions to the Grammar Schema for parsing.

The *parse* function takes a grammar (compatible with schema *S*), and a string, and returns a parse tree labeled with constructors and field bindings interpreted in the context of schema *S*. The *build* function then interprets the parse tree in terms of the schema *S* (provided as a first argument), resulting in an object graph conforming to *S*. Note that *build* is dependently typed: the value of the first argument determines the type of the result, namely *S*. The *format* function realizes the opposite direction: given a grammar compatible with *S* and a value of type *S* it produces a textual representation.

The Ensō parser is implemented as an interpretive variant of the *Generalized LL* (GLL) parsing algorithm [56]. An overview of the Ensō GLL interpreter is provided in Section 5.3. The formatting algorithm recursively searches the Object Grammar to find the relevant productions for rendering objects. This algorithm is described in Section 5.4.

The structure of Object Grammars as described in the Grammar Schema of Figure 7 is not directly supported by any parsing algorithm. For this reason, Object Grammars are first preprocessed to obtain a context-free grammar in the strict, traditional sense.

### *5.2. Modeling Internal Data Structures*

The Grammar Schema extensions are captured by the schema shown in Figure 9. The left column shows how `Pattern` and `Rule` are changed to include additional fields. `Patterns` are enriched with `prev` and `nxt` pointers to link consecutive elements in a sequence. The `Rule` class is extended with an additional super class `Pattern` so that it is allowed as a value for `nxt` pointers.

The new classes `End` and `EpsilonEnd` are used to mark the end of a rule alternative. These are used by the GLL algorithm to pop the parsing stack. Finally the class `Layout` is used to capture the built-in whitespace and comment convention of Ensō Object Grammars. The function of these classes is described in more detail below.

The middle column of Figure 9 describes the structure of the graph-structured stack (`GSS`). GSSs are used in general parsing algorithms to deal with non-determinism in context-free grammars [60]. A GSS node has a pointer to an item (the recognized symbol), a position in the input stream and a set of edges (`Edge`) to other GSS nodes. Each edge is labeled with a (binarized) Shared Package Parse Forest (`SPPF`), which offers an efficient representation of (possibly ambiguous) parse forests.

The structure of SPPFs is shown in the right column of Figure 9. Common to all SPPF nodes is that they span a substring of the input, indicated by the indices `starts` and `ends`. The `type` field captures the syntactic symbol of the node (e.g., a `Rule`). The two subclasses `Node` and `Leaf` represent the result of successfully parsing a fragment of the input. A `Node` corresponds to the result of parsing a context-free symbol. The `kids` field contains one-or-more `Pack` nodes. If a `Node` has more than one child, the node is ambiguous. `Leaf` nodes, on the other hand, capture the result of parsing a terminal symbol. The corresponding text is stored in the `value` field. Finally, `Pack` nodes represent successive parse trees in binarized form via the (optional) `left` field and (required) `right` field. The `pivot` field indicates where the left and right child "meet" in terms of character/token position in the input.

Both GSS and SPPF are essential data structures of the GLL algorithm. By representing these data structures as Ensō models, the generic facilities in Ensō for manipulating, storing and printing models provides powerful capabilities for inspecting the internal data structures of the parsing interpreter. The Grammar Schema extensions in the left column of Figure 9 are used in a preprocessing phase that precedes running the GLL interpreter.

Preprocessing Object Grammars consists of four steps: removing formatting, normalization, adding layout, and sequential linking. Since Object Grammars are represented as object graphs themselves, preprocessing consists of applying a sequence of model transformations to create additional objects and structure so that the grammar can be used for parsing.

The first step is the simplest and consists of removing all formatting directives (Indent, NoSpace and Break) from the Object Grammar. Formatting directives affect neither parsing, nor object graph building and can therefore be discarded.

The second step consists of grammar normalization. This process turns the resulting Object Grammar into one that only consists of rules, non-terminals and terminals. Each rule has one or more alternatives, and each alternative consists of zero or more (non-)terminal symbols (`Call`, `Lit`, `Value`, `Ref`, and `Code`). All other symbols—`Create`, `Field`, `Regular`, and nested `Sequence` and `Alt` symbols—are interpreted as special non-terminals. This means the symbol itself is replaced with a `Call` to a new rule that captures the same syntax as the original. For instance, a regular symbol `X+` is replaced with a call to `IterPlus_`$n$, where $n$ uniquely identifies this very occurrence of `X+`. Additionally, the grammar is extended with a new rule `IterPlus_`$n$ `::= X IterPlus_`$n$ `| X`. The other regular symbols are treated in a similar way.

Field binding (`Field`) and object creation (`Create`) symbols are normalized to rules capturing the syntax of their respective arguments. A link to the original symbol is stored in the `original` (cf. Figure 9) field of the added rules so that this information can be carried over to the parse tree during parsing. As a result, no information about object construction and field binding is lost.

The third preprocessing step consists of inserting a special `Layout` terminal in between every two consecutive symbols. The `Layout` terminal captures the fixed, built-in whitespace and comment convention of Ensō Object Grammars.

Finally, consecutive production elements are linked to one another through `prev` and `nxt` (next) links. The `nxt` pointer of the last element in a sequence always points to an `End` object, which communicates to the parsing algorithm that a production has been fully recognized. For

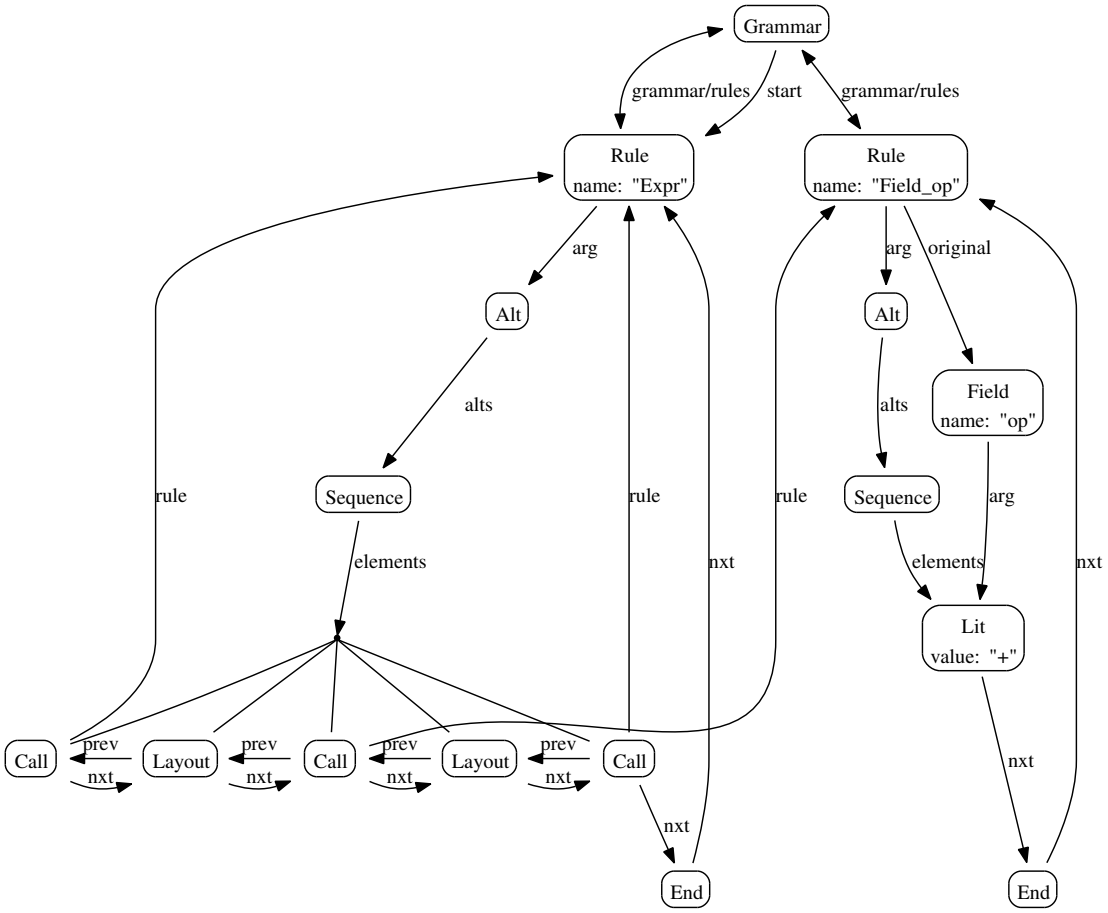



Figure 10: Object graph showing the effect of normalizing and itemizing **start** `Expr Expr ::= Expr op:"+"Expr`

this reason, `End` object have their `nxt` pointers refer to Rule objects. The optional `prev` pointer is used to determine if an element is the first of a sequence. If a production has no elements, a special end object is inserted (`EpsilonEnd`). Linking consecutive production symbols allows the parsing algorithm to use the grammar objects themselves as unique positions in the grammar, and to progressively move through the grammar.

To see the effect of preprocessing Object Grammars, consider the following small grammar:

```
start Expr Expr ::= Expr op:"+" Expr
```

The result of preprocessing could be rendered textually as follows:

```
start Expr
Expr     ::= Expr Layout Field_op Layout Expr
Field_op ::= "+"
```

The resulting version of the grammar consists of plain BNF. What this rendering does not show, however, is the additional structure for running the GLL interpreter and building the object graph. This structure can be seen in the structure of the object graph, as shown in Figure 10. Note how the field binding `op:"+"` is still accessible via the `original` field in the added rule `Field_op`, and note also how the original occurrence of the field binder is replaced by a `Call` to the new rule. The three calls in the production of the `Expr` rule are now interleaved with the `Layout` terminal. Finally, each symbol—including the inserted `Layout` symbols—is linked to its predecessor (if any), and to its successor.

### *5.3. Parsing and Building Object Graphs*

GLL is a general, top-down parsing algorithm, which is both efficient and easy to implement. It supports the complete class of context-free grammars, including grammars with left recursion. In the implementation described here, tokenization of the input stream happens in an on-demand fashion. When a certain token type is expected on the basis of the state of the parser, the scanner is asked to provide this token at the current position of the input stream. If it delivers, parsing continues, otherwise, the current branch is terminated, and other branches in the grammar will be explored. If there are no remaining branches and no parse tree spanning the complete input has been created, a parse error is issued. The result of a successful parse is a (possibly ambiguous) parse forest. The nodes in the parse forest are annotated with grammar patterns (e.g., `Rule`, `Layout` etc.—see Figure 7 and Figure 9).

The generality of GLL supports modularity of syntax definitions. It is well-known that only the full class of context-free grammars is closed under union. Composing two or more grammars will never break the parser. The flip side is that grammar composition might introduce ambiguity. Although ambiguity of context-free grammars is an undecidable property [10], ambiguities are often easy to resolve in practice. Lexical ambiguities are dealt with by the framework using a pragmatic approach. First, a longest-match strategy is applied to all keywords that overlap with the (**`sym`**) identifier syntax. Second, all such keyword literals are always reserved from the identifier syntax, even for composed languages. To force a keyword to be recognized as an identifier, it can be escaped with `\`. Note also that, practically, ambiguity of a grammar is only a problem when actual ambiguous input is parsed. Ambiguous input is always considered to be an error.

The definition of the GLL interpreter is shown in Figure 11 in pseudo code. The left column shows the top-level function `parse`, which receives a grammar (`grm`) and the input (`src`). The global variables `@cu` and `@cn` represent the current GSS stack node and SPPF node respectively. The on-demand scanner is initialized in the `@scan` variable. GLL operates by maintaining a collection of descriptors that represents a work-list (`@todo`).

Before entering the main loop, the algorithm is started by calling the grammar interpreter `eval` (shown on the right) on the start symbol. The main loop acts like a trampoline, taking elements of the work list and dispatching to `eval` until nothing remains to be done. The resulting parse forest, if any, is then found by searching for the SPPF that spans the complete input and is labeled with the start symbol of `grm`.

The actual grammar interpreter `eval` is shown in the right of Figure 11. Recall that every Ensō object is described by a schema. The schema class of an object is accessible through the special field `schema_class`. The `eval` function uses the schema class of the grammar symbol `t` to dispatch to the appropriate action.

The first five cases deal with the terminal symbols `Lit`, `Layout`, `Ref`, `Value` and `Code`. The blocks passed to the scanner methods are only executed if the requested token type is recognized

```
def parse(grm, src)
  @cu = @cn = nil
  @scan = Scanner.new(src)
  @todo = []; @ci = 0;
  eval(grm.start)
  while !@todo.empty? do
    item, @cu, @cn, @ci = @todo.pop
    eval(item)
  end
  pt = node?(grm.start, src.length)
  pt || (raise "Parse error")
end

def terminal(t, i, value)
  cr = get_node_t(@ci, i, t, value)
  @ci = i
  if t.prev.nil? && !t.nxt.End? then
    @cn = cr
  else
    @cn = get_node_p(t.nxt, @cn, cr)
  end
  eval(t.nxt)
end
```

```
def eval(t)
  case t.schema_class.name
  when :Lit then @scan.literal(t.value, @ci) { |i|
     terminal(t, i, t.value)
    }
  when :Layout then @scan.layout(@ci) { |i, ws|
     terminal(t, i, ws)
    }
  when :Ref then @scan.token('sym', @ci) { |i, tk|
     terminal(t, i, tk)
    }
  when :Value then @scan.token(t.kind, @ci) { |i, tk|
     terminal(t, i, tk)
    }
  when :Code then terminal(t, @ci, '')
  when :Rule then t.arg.alts.each { |x|
     add(x.elements[0])
    }
  when :Call then create(t.nxt); eval(t.rule)
  when :End then pop
  when :EpsilonEnd then
    cr = get_node_t(t, @ci)
    @cn = get_node_p(t.nxt, @cn, cr)
    pop
  end
end
```

Figure 11: Pseudo Ruby code of the GLL interpreter

in the input at position `@ci`. Otherwise, control returns to the main loop in `parse`. If the terminal is successfully recognized, the routine `terminal` (shown on the left) creates the necessary SPPF nodes, using the helper routines `get_node_t` (`Leaf`) and `get_node_p` (`Node`). After that, parsing immediately continues with the next element in the sequence (`eval(t.nxt)`).

The last four cases (`Rule`, `Call`, `End` and `EpsilonEnd`) invoke additional GLL helper routines. First, `add` schedules new work items on the work list. This procedure is called when a production is encountered. Second, `create` pushes new GSS nodes which function as "return points" after a non-terminal has been recognized. Third, the `pop` routine pops the GSS, possibly creating additional SPPF nodes and scheduling additional work items. This routine is called if the end of a production is reached (`End`, or `EpsilonEnd` symbols). Recall that `EpsilonEnd` captures an empty production. For this reason an empty SPPF node is recorded using `get_node_t` and `get_node_p`.

The precise semantics of `get_node_t`, `get_node_p`, `add`, `create`, and `pop` is outside of the scope of this paper. For more information we refer to the work by Scott and Johnstone [56] which contains full definitions of these five routines. One important aspect however, is that GSS, SPPF and Pack nodes should be *shared*. This means that a GSS is only created if no GSS with the same `item` and position (`pos`) already exists. Similarly, an SPPF is only created if no SPPF with the same `starts` and `ends` and `type` values already exists. Finally, Pack nodes are shared based on the values of the `parent`, `type` and `pivot` fields.

```
def build(pt, ob=nil, f=false, vs=[], ps=[])
  case pt.schema_class.name
  when :Node then
    amb_error(pt) if is_amb?(pt)
    build_node(pt.type, pt, ob, f, vs, ps)
  when :Leaf then
    build_leaf(pt.type, pt, ob, f, vs, ps)
  when :Pack then
    build(pt.left, ob, false, vs, ps)
    build(pt.right, ob, !pt.left.nil? , vs, ps)
  end
end

def fixup(root, fixes)
  begin
    later = []; change = false
    fixes.each do |path, obj, fld|
      x = path.deref(root, obj)
      if x then # the path can be resolved
        update(obj, fld, x)
        change = true
      else # if not, try it later
        later << [path, obj, fld]
      end
    end
    fixes = later
  end while change
  raise "Fix-up error" unless later.empty?
end
```

```
def build_node(s, pt, ob, f, vs, ps)
  case s.schema_class.name
  when :Rule then
    if s.original then
      build_node(s.original, pt, ob, f, vs, ps)
    else
      build(pt.kids[0], ob, f, vs, ps)
    end
  when :Create then
    build(pt.kids[0], ob=@fact.make(s.name))
  when :Field then
    build(pt.kids[0], ob, true, vs=[], ps=[])
    vs.each { |v| update(ob, s.name, v) }
    ps.each { |p| @fixes << [p, ob, s.name] }
  else
    build(pt.kids[0], ob, f, vs, ps)
  end
end

def build_leaf(s, pt, ob, f, vs, ps)
  case s
  when :Lit then vs << pt.value if f
  when :Value then
    vs << convert(pt.value, s.kind)
  when :Ref then
    ps << subst_it(pt.value, s.path)
  when :Code then assert(s.code, ob)
  end
end
```

Figure 12: Pseudo Ruby code to turn parse trees into object graphs. The entry function is `build` (upper left).

To implement sharing of GSS and SPPF nodes, Ensō leverages the fact that all data is managed by schema interpreters [41]. This means that when and how objects are created and accessed can be customized to a great extent. In this particular instance, we have implemented a custom data manager (extending the default one) which employs a technique similar to hash-consing [27] to realize sharing. If a constructor is called with the same arguments for a second time, the factory returns a previously allocated object. As a result, the cross-cutting aspect of sharing is now defined at a single place.

If parsing is successful, the *build* function creates the object-graph from the concrete syntax tree returned from the *parse* function. This happens in two steps. First, the spine of the object graph is created using the function `build` shown in Figure 12. The first argument to `build` is the syntax tree, `ob` represents the "current" object, and `f` indicates if field assignment can be performed. Finally, `vs` and `ps` collect values and paths respectively. The `build` procedure recursively traverses the syntax tree and depending on whether a node is a `Node` or a `Leaf` calls into either `build_node` or `build_leaf` (shown at the right of Figure 12). If the current node `pt` is `Pack` node, both the left node (if any) and the right node are built recursively.

The `build_node` function creates objects and assign fields based on the `type` annotation of the

SPPF. If the rule is the result of normalization, `build_node` is called recursively with the same `pt`, but with a different label, `s.original`. This ensures that normalized `Create` and `Field` are taken into account. For other rules, `build_node` simply continues with the children. If the `kids` field of a `Node` SPPF contains more than one elements, an ambiguity error is raised in `build`. Hence, only the first child (`pt.kids[0]`) is passed to recursive invocations of `build_node`.

For constructor directives `Create`, the factory object `@fact` is asked to create an object of the right class. The created object becomes the new current object when recursing down the tree. In the case for `Field` nodes, the values collected in `vs` are directly assigned to the corresponding fields in the current object. The paths `ps` are recorded as "fixes" to the current object for the current field in the global variable `@fixes`; these fixes are applied later to create cross-links.

The `build_leaf` routine deals with terminal symbols. Both `Lit` and `Value` values are simply added to the collection of values `vs`. `Values` are first converted to the expected type; the value of a literal is recorded literally, but only when the node is directly below a field binder (i.e., `f` is true). When a reference is encountered (`Ref`) the special keyword **`it`** is substituted for the name that has been parsed, and the resulting path expression is added to `ps`. Finally, `Code` predicates are asserted in the context of the current object using an expression interpreter `assert`. The fields of `ob` are assigned so that the expression `s.code` becomes true.

In the second step, the path-based references are resolved in an iterative fix-point process. This is shown in the bottom, right-hand side of Figure 12. The fix-point process ensures that dependencies between references are dynamically discovered. If in the end some of the paths could not be resolved—for instance because of a cyclic dependency—an error is produced.

### *5.4. Formatting*

Formatting works by matching objects againsts constructor and field specifications in an Object Grammar. In essence, the formatter searches for a rendering that is compatible with the object graph. When the class in a constructor directive matches the class of the object being formatted, the object is formatted using the body of the production. If formatting fails when recursing through the grammar, the formatter backtracks to select a different production. If no suitable alternatives can be found, an error is raised.

Figure 13 shows the (slightly simplified) algorithm for rendering an object graph to text. The `format` function receives a grammar symbol `s`, an object stream `os` and an output stream `out` (initialized as an empty list). The function returns a boolean indicating whether formatting succeeded or not. The object stream represents a cursor over the object graph, containing a single (`Stream1`) or a collection of objects (`StreamN`). After formatting, the output stream will consist of sequence of strings and formatting objects (`NoSpace`, `Ident`, and `Break`). This result is then rendered to text in a straightforward way.

For rules and non-terminal calls, `format` recurses to the `arg` and `rule` fields, respectively. The formatting symbols are simply added to the output stream; they will be interpreted when the output stream is rendered to text. `Code` objects do not contribute any output, but might indicate failure, namely when the code expression `s.expr` evaluates to false in the context of the current object. When encountering an `Alt` symbol, `format` tries every alternative with a clean output stream `sub`, and on first success appends the result to `out`, and returns true. Sequencing is dual to alternation: in this case each sub element must be formatted successfully for the formatting of the `Sequence` symbol to be successful. If this is the case, all output generated by the sub elements (`total`) is appended to `out`.

An object can be rendered if it matches up to a `Create` symbol with the same name. In that case, the object cursor (`os`) is moved to the next element, and the current object is formatted

```
def format(s, os, out=[])
  case s.schema_class.name
  when :Rule then
    return format(s.arg, os, out)
  when :Call then
   return format(s.rule, os, out)
  when :NoSpace, :Indent, :Break then
    out << s
    return true
  when :Code then
    return eval_expr(s.expr, os.current)
  when :Alt then s.alts.each { |x|
     if format(x, os.copy, sub = []) then
       out += sub; return true
     end
    }
  when :Sequence then
    total = []; ok = true
    s.elements.each { |x|
     ok &= format(x, os, sub = [])
     total += sub
    }
    out += total if ok
    return ok
```

```
  # continued from previous column...
  when :Create then
    o = os.current
    if o && o.schema_class.name == s.name then
      os.next
      return format(s.arg, Stream1.new(o), out)
    end
  when :Field then # [literal case omitted]
    o = os.current
    fld = o.schema_class.all_fields[s.name]
    return format(s.arg, fld.many ?
                StreamN.new(o[s.name]) :
                Stream1.new(o[s.name]), out)
  when :Lit then
    out << s.value; return true
  when :Value then
    return format_value(s.kind, os.current, out)
  when :Ref then
    out << solve_path(s.path, os.current)
    return true
  end
  return false
end
```

Figure 13: Pseudo Ruby code of the object graph formatting algorithm

using the argument of a `Create` symbol. The formatting of a `Field` consists of retrieving the field's value on the current object, and formatting it with a fresh object cursor. For the sake of brevity, we have omitted the code that deals with field arguments that consist of a literal. In that case, the value stored in the current object should be equal to the value of the literal itself.

The value of a literal (`Lit`) is simply added to the output. For `Value` terminals, however, the helper function `format_value` uses the `kind` field to apply necessary escaping and quoting of strings. Finally, in order to format cross-links in the object graph, the path of `Ref` is used to find the value of `it`. So, whereas the `fixup` process searches for objects based on a path, and identifier and an object graph, here the target object is given from the start, and the path is used to find the textual name to output.

The cases for `Regular` symbols have been omitted from Figure 13. It is however easy to see that they can be handled through normalization to combinations of `Alt` and `Sequence`, similar to the normalization described in Section 5.3.

The alternatives of a rule are represented using ordered collections because the `Alt` class (see Figure 7) does not have a key field. For parsing the ordering is irrelevant: in GLL all alternatives are explored in "parallel". However, as is clear from the code in Figure 13, the formatter explores the alternatives in the order of declaration. If a rule is recursive without providing sufficient constraints on an alternative to actually recurse deeper into the object graph, the formatter might not terminate.

Consider the following grammar:

```
Exp ::= "(" Exp ")"
      | [Var] name:sym
      | ...
```

In this case the formatter will keep on trying the first bracketing alternative because it never fails on an expression object. As result, the formatter will forever try to surround the expression to be rendered with parentheses without ever rendering the expression itself.

In practice this situation is avoided by placing such alternatives as the last one in the list, so that parentheses are only added when really needed. Nevertheless, non-termination can always resurface after composing such a grammar with another one. A completely general solution will explore the alternatives based on a kind of specificity ordering, where the alternatives imposing the strongest constraints are explored first.

### *5.5. Merging*

In Ensō, composition of grammars and schemas are both accomplished using the same generic merge operator $\cdot \lhd \cdot$. This operator can be characterized as an overriding union where conflicts are resolved in favor of the second argument. Since a language is defined by its schema and grammar, the composition of a base language $B$ with an extension $E$ is given by composing grammars $G_B \lhd G_E$ and schemas $S_B \lhd S_E$.

The algorithm implementing $\lhd$ is shown in pseudo Ruby code in Figure 14. There are two passes in the merge algorithm. In the first pass, `build` traverses the spine of the object graph `o1` to create any new object required. If `build` encounters an object in `o2` but none at the same location on the spine in `o1`, it creates a new copy of that object and attaches it to the graph of `o1`. Primitive fields from `o1` are always overridden by the same fields of `o2`, allowing the extension to modify the original language. Pairs of objects are merged by merging the values of each field. Collections are merged pair-wise according to their keys; `outer_join` is a relational join of two

```ruby
def merge_into(type, o1, o2)
  build(type, o1, o2, memo = {})
  link(type, true, o1, o2, memo)
end

def build(type, a, b, memo)
  return if b.nil?
  memo[b] = new = a || type.new
  type.fields.each do |fld|
    ax = a && a[fld.name]
    bx = b[fld.name]
    if fld.type.Primitive? then
      new[fld.name] = bx
    elsif fld.spine
      if !fld.many
        build(fld.type, ax, bx, memo)
      else
        ax.outer_join(bx) do |ai, bi|
          build(fld.type, ai, bi, memo)
        end
      end
    end
  end
end
```

```ruby
def link(type, spine, a, b, memo)
  return a if b.nil?
  new = memo[b]
  return new if !spine
  type.fields.each do |fld|
    ax = a && a[fld.name]
    bx = b[fld.name]
    next if fld.type.Primitive?
    if !fld.many? then
      val = link(fld.type, fld.spine, ax, bx, memo)
      new[fld.name] = val
    else
      ax.outer_join(bx) do |ai, bi|
        x = link(fld.type, fld.spine, ai, bi, memo)
        unless new[fld.name].include?(x)
          new[fld.name] << x
        end
      end
    end
  end
  return new
end
```

Figure 14: Pseudo Ruby code for the generic $\triangleleft$ operator

collections, matching up all pairs of items with equivalent keys and pairing up the remaining items with `nil`. For instance, applying merge to two Object Grammars combines the alternatives of rules with the same name in both grammars. If the collection is a list (i.e., containing objects without a key), the elements are concatenated. An example of this is merging two collections of rule alternatives; `Alt` objects do not have a key (cf. Figure 7), so the result of merging is the concatenation of both collections.

The first pass also establishes a mapping `memo`, between each object in `o2` and the corresponding object in the same spine location in `o1`. This mapping is used in the second phase, where non-spine fields—those without the `!` modifier—are made to point to their new locations. The object graph is once again traversed along the spine, but this time `link` looks up `memo` for each non-spine field in order to find the updated target object.

Note how the algorithm exploits the fact that every Ensō model is described by a schema, which is again a model. For instance the main loop of `build` iterates over the fields of the `type` parameter, which is an instance of the class `Type` in the Schema Schema (Figure 5). Each field `fld` is queried for its name, type and multiplicity to direct the algorithm. In fact, merge, and other such operations, may all be considered to be interpreters over some model.

The `outer_join` call in Figure 14 matches objects in two keyed collections based on their key. This implies that objects in such collections live in the same name space. Often this is what is actually desired, for instance to concatenate lists of productions of two rules with the same name, or to union field sets if two classes have the same name. On the other hand, there is currently no

mechanism for qualifying names to avoid name clashes. To cater for the scenario where named elements should stay in their own name spaces, a generic *rename* operation can be used, similar to symbol renaming in SDF [64]. For instance, in the case of a grammar, this operation can be used to selectively rename rules or referenced classes in constructors. If referenced classes are renamed, a corresponding rename is also needed on the schema.

### *5.6. Bootstrapping*

Since Ensō is self-describing, parsing and building the four core models of Section 4 require a bootstrapping process. Loading a model from a file requires a grammar for that particular type of model. We also need a schema so that parse trees produced by the parser can be instantiated. All objects in an object graph have an "instance of" pointer to the schema class they are instantiated from (`schema_class`). Unfortunately, the four core models, are mutually dependent. For instance, Grammar Schema requires Schema Grammar and Schema Schema, and Schema Grammar requires Grammar Schema and Grammar Grammar, etc. The bootstrapping process ensures that the circular dependency of the four core models is broken, and that the `schema_class` pointers eventually point to classes of the right schema. In particular, the schema of schemas conforms to itself, so its `schema_class` points to itself.

Let $x_L$ designate the contents of a file `x.L`, where the file extension `L` indicates the language of the model, then the following recursive equation captures how Ensō models are loaded:

$$load(x_L) = load_{gs}(x_L, load(L_{grammar}), load(L_{schema})) \tag{4}$$

Where $load_{gs}$ is defined as:

$$load_{gs}(x_L, g, s) = build(parse(g, x_L), factory(s)) \tag{5}$$

Bootstrapping the Ensō system requires computing the fixed point of Equation 4. For instance, to load a state machine model `doors.statemachine`, we first try to load the `statemachine.grammar` and `statemachine.schema`. Then, the input is parsed using the grammar and the result is passed to a factory initialized with the schema. But of course, loading `statemachine.grammar`, requires loading `grammar.grammar` and `grammar.schema`. In turn, `grammar.grammar` can only be loaded if `grammar.grammar` and `grammar.schema` are available.... Furthermore, loading `statemachine.schema`, requires `schema.grammar` and `schema.schema`. And to load `schema.grammar`... And so on. To resolve this infinite regression, the four core models of Figure 8 are loaded through an intricate bootstrap process, which is visualized in Figure 15.

The diagram can be read from left to right. The solid arrows indicate *loading* steps. Each step is labeled with a number indicating order of occurrence. The dashed arrows represent the relation "described by" or "instance of". The nodes in the diagram with italic labels indicate object graphs; the other nodes represent files. A box around a grammar and one or more files indicates that those files are parsed using that grammar. Below we briefly describe each successive step.

1. A bootstrap version of schema schema is loaded from a JSON [13] file using a mock SchemaSchema that is implemented in plain Ruby. The loaded SchemaSchema is an instance of the mock SchemaSchema.
2. Since we now have a proper SchemaSchema, we can create schemas. This is used when loading the grammar schema from JSON.
3. The GrammarSchema can be used to create grammars. So, this time, we load the bootstrap version of the grammar of grammars. We are now in the situation that we can parse grammars.

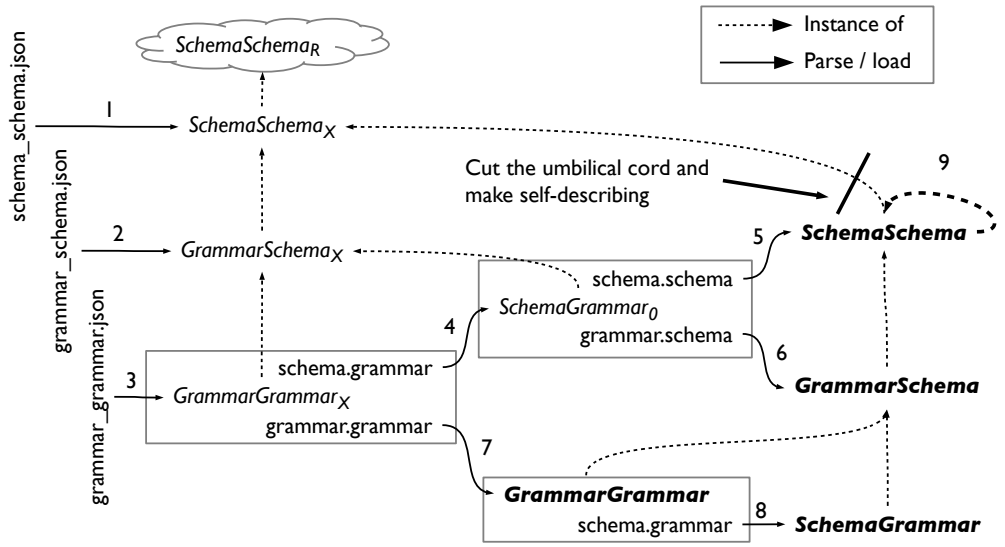



Figure 15: Visualization of the Ensō bootstrap process

4. The first grammar that is loaded by parsing is SchemaGrammar. Its instance-of pointer points to the GrammarSchema obtained in step 2. We can now load schemas from file.
5. The real SchemaSchema is loaded from file. Its instance-of pointer, however, still points to the bootstrap SchemaSchema obtained in step 1.
6. The real GrammarSchema is loaded from file. It will be an instance of the real SchemaSchema of the previous step.
7. Because the real GrammarSchema is now available, and we have a grammar to parse grammars, we can load the real GrammarGrammar. It will be an instance of GrammarSchema loaded in step 6.
8. Since the previous SchemaGrammar pointed to the bootstrap GrammarSchema of step 2, the real SchemaGrammar is loaded from file to ensure that it is an instance of the real GrammarSchema (step 6).
9. Finally, the link from SchemaSchema (step 5) to the bootstrap SchemaSchema (step 1) is severed; all instance-of pointers are made self-referencing.

After the bootstrap is complete, the loaded models are cached, so that loading of additional models just involves the four core models obtained in steps 5, 6, 7, and 8.

For the sake of presentation we have omitted the way Object Grammar composition affects the bootstrap process. Since both grammars and schemas require an expression language, the bootstrap process is actually more complex than described here. When the expression language is not yet available, computed fields in schemas, and predicates in grammars are coded directly in Ruby. As soon as the four core models are available, the expression grammar and schema are loaded and the bootstrap process is restarted. Finally, the extensions to the Grammar Schema needed for parsing and normalization (Figure 9) are literally included in the bootstrap version. In the second round they are merged into the Grammar Schema of Figure 7.

## 6. Evaluation

### *6.1. Introduction*

The goal of Ensō is the definition, composition and interpretation of DSLs. The textual syntax of these DSLs is defined using Object Grammars. Object Grammars are used to load DSL programs into the system. The Ensō system currently consists of a number of such DSLs. In this section we elaborate how well Object Grammars live up to their promise. In particular we aim to show the following:

- Practicality: Object Grammars can be used to define practical languages.
- Variety: the formalism allows different kinds of languages (e.g., tree- and graph-like languages), to be defined in a concise way.
- Reuse: the composition of Object Grammars may lead to significant reuse across languages.

Recall that the goal of Object Grammars is not to be able to define existing, general-purpose language such as Java or COBOL. Unlike other systems that do have this goal, such as Rascal [35], Object Grammars are targeted solely at defining new DSLs. The cross-linking feature of Object Grammars is also not to be considered as a substitute for full name-analysis of such languages (which is, for instance, the key goal of NaBL [36]).

As we hope to have shown in the course of this paper, Object Grammars can be used to define practical languages. Both the foundational schema and grammar languages are defined using Object Grammars. In addition, the Ensō system currently features a language for defining GUIs (Stencil), security policies (Auth) and Web applications (Web). The Stencil and Web languages are languages to transform arbitrary Ensō models to a GUI resp. Web interface. Of the current set of languages in Ensō, these languages are most like a ordinary programming language in that they feature control-flow statements and expressions. The Grammar, Schema and Auth languages are much more declarative languages. In the grammars for Grammar and Schema the path-based references are essential in resolving cross references. Furthermore, Object Grammars were used in the Ensō submission to the Language Workbench Challenge 2012 (LWC'12) [40]. This involved two inherently graph-like languages. Below we first review the languages provided with Ensō, how they are composed and what benefits in terms of reuse had been gained. Second, we discuss the LWC'12 case-study in some more detail.

### *6.2. Composition in Ensō*

Many of the current set of languages in Ensō are defined by composing two or more language modules. Figure 16 shows how Ensō languages are related with respect to language composition. Each edge in the diagram represents an invocation of $\lhd$. The arrow points in the direction of the result. For instance, the Stencil and Web languages are, independently, merged into the Command language. As a result both Stencil and Web include, and possibly override and/or extend the Command language. If a language reuses or extends multiple other languages, the merge operator is applied in sequence. For instance, Command is first merged into Web, and then XML is merged into the result.

The core languages in Ensō include both the Schema and Grammar languages, as well as Stencil, a language to define graphical model editors, and Web a language for Web applications. Additionally, Ensō features a small set of library languages that are not deployed independently

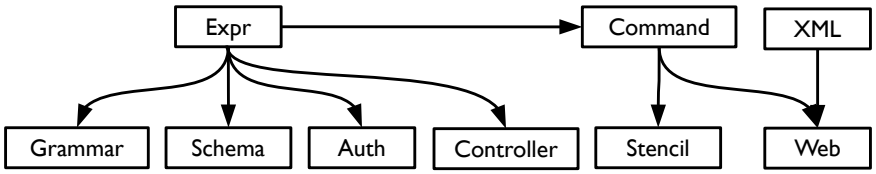



Figure 16: Language composition in Ensō. Each arrow $A \rightarrow B$ indicates an invocation of $B \lhd A$.

| **Language** | Schema | Grammar | Interpreter |
|---|---|---|---|
| Grammar | 55 | 31 | 1394 |
| Schema | 31 | 20 | 744 |
| Stencil | 68 | 28 | 1387 |
| Web | 79 | 43 | 885 |
| Auth | 28 | 16 | 276 |
| Piping | 80 | 22 | 306 |
| Controller | 26 | 14 | 155 |
| Command | 36 | 24 | 114 |
| Expr | 45 | 30 | 113 |
| XML | 10 | 6 | 47 |

(a)

| Reuse Percentages | | | |
|---|---|---|---|
| **Language** | Schema | Grammar | Interpreter |
| Grammar | 45% | 49% | 20% |
| Schema | 59% | 60% | 12% |
| Stencil | 54% | 76% | 20% |
| Web | 51% | 55% | 31% |
| Auth | 62% | 65% | 25% |
| Piping | 36% | 58% | 27% |
| Controller | 64% | 45% | 42% |

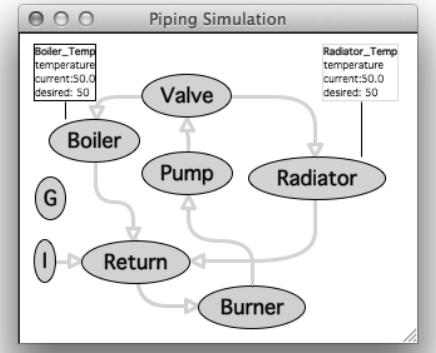
(b)

Table 2: SLOC count (a) and reuse percentages (b) for schemas, grammars and interpreters of the languages currently in Ensō

but reused in other languages. An example of a library language is Expr, an expression language with operators, variables and primitive values. It is, for instance, reused in Grammar for predicates and paths and in Schema for computed fields. Command is a control-flow language that captures loops, conditional statements and functions. The Command language reuses the Expr language for the guards in loops and conditional statements.

The composition with the Expr language is an example of language reuse. The language is reused as a black box, without modification. The composition of Command with Stencil and Web is an example of language mixin. Stencil is created by adding language constructs for user-interface widgets, lines, and shapes to the Command language as valid primitives. The Command language can now be used to create diagrams. A similar extension is realized in the Web language: here a language for XML element structure is mixed with the statement language of Web. The extension works in both directions: XML elements are valid statements, statements are valid XML content. The Piping and Controller languages are from a domain-specific modeling case-study in the domain of piping and instrumentation for the Language Workbench Challenge 2012 [19, 40]. Figure 16 only shows the Controller part which reuses Expr.

An overview of the number of non-empty, non-comment source lines of code (SLOC) is shown in Table 2(a). We show the number for the full languages in Ensō as well as the reused language modules (Command, Expr and XML). A language consists of a schema, a grammar and an interpreter. The interpreters are all implemented in Ruby. Table 2(b) shows the reuse percentage for each language [25]. This percentage is computed as $100 \times \#SLOC_{\text{reused}}/\#SLOC_{\text{total}}$. Which languages are reused in each case can be seen from Figure 16. As can be seen from this




```
I: source water
G: source \gas

Burner: burner in=Return gas=G
Pump: pump [ l:10.0 d:0.1 ] in=Burner
Valve: splitter in=Pump
Boiler: vessel in=Valve.left
Radiator: radiator in=Valve.right
Return = Boiler + Radiator + I + G

control Radiator_Temp:
  temperature (Radiator)
control Boiler_Temp:
  temperature (Boiler)
```

Figure 17: Diagram editor (left) showing a textual piping model of a simple heating system (right)

table, the amount of reuse in schemas and grammars is consistently high. It shows that the merge operator is powerful enough to combine realistic languages in a variety of ways, with actual payoff in terms of reuse.

### *6.3. Case-Study: Piping and Instrumentation*

To demonstrate the capabilities of the Ensō system, we have performed a case-study in the domain of piping and instrumentation [70] for the Language Workbench Challenge 2012 [40]. The resulting application can be used to simulate models of heating systems. There are two parts to this system: a language for defining the piping circuit, and a controller language to manage its behavior.

Figure 17 shows a screen shot of a running diagram editor (left) of an example piping model (right). The Object Grammar for piping models is shown in Figure 18. The diagram editor is constructed using the graphical model editor language Stencil. Internally, the piping language uses automatic reference resolving of Object Grammars to connect pipes to various components, such as valves, pumps and heaters. This can be observed in the `Sensor` and `Conn` grammar rules.

The dynamic behavior of a piping model is determined by a separate controller language, which is a state machine language similar to but more extensive than the language of Figure 2. Its Object Grammar is shown in Figure 19. It reuses the Expr language to represent enabling conditions on transitions. Transitions are connected to states using path-based references.

The schema of the piping layout language is enriched with additional fields and classes to represent dynamic state of a simulation (temperature, pressure and flow). This is similar to how the Grammar Schema is extended for parsing (Section 5.2). The Stencil model which maps the piping model to the interactive visualization exploits this information to reflect temperature changes through the animation of coloring of nodes and edges (Figure 17).

Figure 20 shows a diagram editor for the state machine controlling the piping model of Fig 17. The current state is highlighted in dark. Since both windows in Figure17 and Figure 20 are actually diagram-based editors, the models can be changed directly while the simulation is running.

```
start System
System  ::= [System] elements:Element* sensors:Sensor*
Element ::= [Source]   name:sym ":" "source" outputs:Pipe kind:sym
         |  [Exhaust]  name:sym ":" "exhaust" input:Pipe  name:sym
         |  [Vessel]   name:sym ":" "vessel"  outputs:Pipe Inputs
         |  [Valve]    name:sym ":" "valve" outputs:Pipe Inputs
         |  [Splitter] name:sym ":" "splitter" outputs:(Pipe Pipe) Inputs
         |  [Pump]     name:sym ":" "pump" outputs:Pipe Inputs
         |  [Radiator] name:sym ":" "radiator" outputs:Pipe Inputs
         |  [Joint]    name:sym outputs:Pipe "=" inputs:Conn* @"+"
         |  [Burner]   name:sym ":" "burner" Inputs Gas outputs:Pipe
         |  [Room]     name:sym ":" "room"
Sensor  ::= [Sensor] ("sensor"  {controllable==false} |"control" {controllable==true})
              name:sym ":" kind:sym
              "("  (attach:<root.elements[it]> | attach:Conn)  ")"
Pipe    ::= [Pipe] { length == 0.0 and diameter == 0.0}
         |  [Pipe] "[" ("l" ":" length:real)? ("d" ":" diameter:real)? "]"
Inputs  ::= "in" "=" inputs:Conn
Gas     ::= "gas" "=" gas:Conn
Conn    ::= <root.elements[it].outputs[0]>
         | <root.elements[it].outputs[0]> "." "left"
         | <root.elements[it].outputs[1]> "." "right"
         | <root.elements[it].input> "." "input"
```

Figure 18: Object Grammar for piping layout models.

```
start Ctl
Ctl        ::= [Controller] "start" initial:<root.states[it]>
               / globals:Assign*@/
               / states:State*@/
               constraints:Constraint*@/
State      ::= [State] "state" name:sym ":" transitions:Transition*@/ commands:Action*@/
Action     ::= Assign | Splitter
Assign     ::= [Assign] var:Expr "=" val:Expr
            |  [Assign] "raise" var:Expr "by" val:Expr
            |  [Assign] "lower" var:Expr "by" val:Expr
Splitter   ::= [TurnSplitter] "turn" "splitter" splitter:sym
                     ( "left"   {percent == 0.0} |
                       "center" {percent == 0.5} |
                       "right"  {percent == 1.0} )
Transition ::= [Transition] "if" guard:Expr "goto" target:<root.states[it]>
Constraint ::= [Constraint] "test" cond:Expr "do" action:Action
abstract Expr
```

Figure 19: Object Grammar for piping controllers.

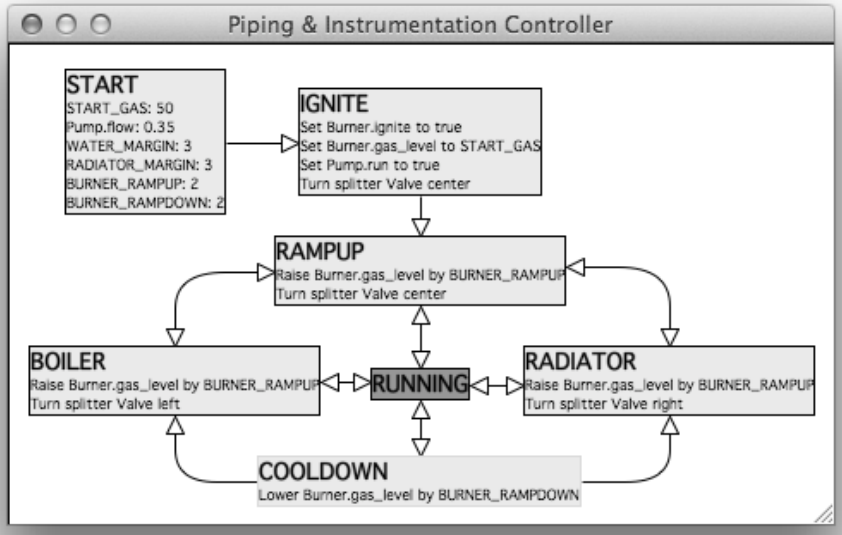



Figure 20: Diagram editor for a controller for simulating a simple heating system

| System | Ref. | Bidirectional | Compositional | Resolving | Asynchronous | Self-describing |
|---|---|---|---|---|---|---|
| TCS | [34] | ◐ | ○ | ● | ● | ● |
| Xtext | [20] | ● | ○ | ● | ● | ○ |
| EMFText | [14, 29] | ● | ○ | ● | ● | ● |
| MontiCore | [37] | ○ | ○ | ● | ○ | ○ |
| TCSSL | [23] | ◐ | ○ | ● | ○ | ● |
| Yacc | [31] | ○ | ○ | ◐ | ● | ○ |
| SDF | [64] | ● | ● | ○ | ○ | ● |
| Ensō | | ● | ● | ● | ● | ● |

Table 3: Positioning syntax definition frameworks.

The piping and instrumentation case-study shows the utility of both automatic reference resolution, and composition of object grammars. Both the language for specifying piping layouts and the language for specifying controllers require cross-links. Furthermore, the controller language reuses the Expr language for its enabling conditions. The piping layout models are enriched with additional structure for maintaining dynamic state using the generic merge operator. The complete system is built on a total of six (sub-)languages: the domain-specific Piping and Controller languages, and the reusable Stencil, Grammar, Schema, and Expr languages.

## 7. Related Work

### *7.1. Grammars and Models*

The subject of bridging modelware and grammarware is not new [1, 71]. In the recent past, numerous approaches to mapping text to models and vice versa have been proposed [20, 22, 29, 34, 37, 44, 45]. Common to many of these languages is that references are resolved using globally unique, or hierarchically scoped names. Such names can be opaque Unique Universal Identifiers (UUIDs) to uniquely identify model elements or key attributes of the elements themselves [26]. The main difference between these approaches and Object Grammars is that Object Grammars replace the name-based strategy by allowing arbitrary paths through the model to find a referenced object. This facilitates mappings that require non-global or non-hierarchical scoping rules.

A feature-based comparison of representative systems is shown in Table 3. The table includes Yacc [31] as the archetypical action-based parser generator. SDF [64] is included to represent tree-based grammar formalisms. Both can be considered to be at opposite ends of a spectrum of syntax definition formalisms, from strictly imperative, to purely declarative. Below we discuss the model-based systems in more detail.

The Textual Concrete Syntax (TCS) language supports deserialization and serialization of graph-structured models [34]. Field binders can be annotated with `{refersTo =` ⟨*name*⟩`}`, which binds the field to the object of the field's class with the field ⟨*name*⟩ having the value of the parsed token. Rules can furthermore be annotated with `addToContext` to add it to the, possibly nested, symbol table. The symbol table is built after the complete source has been parsed to allow forward references. Only simple references to in-scope entities are allowed, however. Path-based references of Object Grammars allow more complex reference resolution, possibly across nested scopes. TCS aims to have preliminary support for pretty printing directives to control nesting and indentation, spacing and custom separators. However, these features seem to be unimplemented.

Xtext is an advanced language workbench for textual DSL development [20]. The grammar formalism is restricted form of ANTLR so that both deserialization and serialization is supported. Xtext supports name-based referencing. To customize the name lookup semantics Xtext provides a Scoping API in Java. Apart from the use of simple names, Xtext differs from Object Grammars in that, by default, linking to target objects is performed lazily. Again, this can be customized by implementing the appropriate interfaces. Xtext supports a limited form of modularity through grammar mixins. For lexical syntax Xtext provides a standard set of terminal definitions such as INT and STRING, which are available for reuse.

EMFText is an Ecore based formalism similar to Xtext grammars [14, 29]. EMFText, however, supports accurate unparsing of models that have been parsed. For models that have been created in memory or have been modified after parsing, formatting can be controlled by pretty printing hints similar to the `.` and `/` symbols presented in this paper. The grammar symbol `#`*n* forces printing of the *n* spaces. Similarly, `!`*n* is used for printing a line-break, followed by *n* indentation steps.

In the MontiCore system both meta-model (schema) and grammar are described in a single formalism [37]. This means that the non-terminals of the grammar introduces classes and syntactic categories at the same time. Grammar alternatives are declared by non-terminal "inheritance". As a result, the defined schema is directly tied to the syntactic structure of the grammar. The formalism supports the specification of associations and how they are established in separate sections. The default resolution strategy assumes file-wide unique identifiers, or syntactically hierarchical namespaces. This can be customized by programming if needed.

The Textual Concrete Syntax Specification Language (TCSSL) is another formalism to make grammars meta-model-aware [23]. It features three kinds of syntax rules: CreationRules which function like our `[Create]` annotations,—SeekRules, which look for existing objects satisfying an identifying criterion,—and SingletonRules, which are like CreationRules, but only create a new object if there is no existing object satisfying a specified criterion. The queries used in SeekRules seem more powerful than simple, name-based resolution; it is however unclear from the paper how they are applied for complex scenarios. TCSSL furthermore allows code fragments enclosed in double angular brackets (`<<>>`) but it is unclear how this affects model-to-text formatting.

An interesting point in the spectrum between Yacc and SDF is obtained by observing that parser and pretty printer are in fact isomorphic. In [52], the authors exploit this fact to present a generic, polymorphic interface for specifying syntactic descriptions. Such descriptions capture enough information so that the interface can be implemented by both parsing and formatting algorithms. As a result, a single specification is sufficient to get a parser and formatter for free. Although this framework is AST-based, the resulting descriptions are quite similar in aim to Object Grammars.

### *7.2. Language Composition*

Modular language development is an active area of research. This includes work on modular extension of DSLs and modeling languages [42, 62, 66, 67], extensible compiler construction [5, 17], modular composition of lexers [11] and parsers [9, 55], modular name analysis [16] and modular language embedding [53]. For an overview the composition capabilities of various systems we refer to [18].

Object Grammars support a powerful form of language composition through the generic merge operation ($\lhd$) applied in tandem to both grammars and schemas. The merge operator

covers language extension and unification as discussed in [18]. In essence, merge captures an advanced form of inheritance similar to feature composition [2, 3]. However, merge currently applies only syntactic and semantic *structure*. To achieve the same level of compositionality at the level of *behavior*, i.e. interpreters, is an important direction for further research. A promising approach for realizing interpreter composition is the recent work on feature-oriented programming using Object Algebras [47].

Composition of grammars is well-studied topic in computer science. It is well-known that only the full class of context-free grammars (CFG) is closed under composition. Composing to grammars in subclasses of context-free grammars, such as LL(k) or LR(k), is not guaranteed to produce a grammar that is again in the subclass. This fact motivated the choice of using general parsing as the basis of Object Grammars. Parsing Expression Grammars (PEGs) [24] are also closed under union and, moreover, never lead to ambiguities. However, unlike CFGs, the alternatives in a grammar rule are ordered and, consequently, the composition of alternatives when combining grammars is ordered as well. This can lead unexpected results when parsing using the composed grammar. For instance, composing PEGs `A ::= a` and `A ::= a B` produces the grammar `A ::= a / a B` (where `/` indicates ordered choice). The alternative `a B` has now become unreachable because PEG parsing does not backtrack over rule alternatives: as soon as the first `a` is recognized, the second alternative is never considered. Another drawback is that PEGs generally do not support left-recursion (see however, [61, 69]).

### *7.3. Self-Description*

Self-description is a well-known concept in different areas of programming languages and software engineering. In model-driven engineering, for instance, it is commonly assumed that the meta model of meta models, the "meta meta model", conforms to itself [8, 32]. This is similar to how the Schema Schema describes itself. Self-describing grammar formalisms are almost as old as BNF itself. Most parser generators employ formalisms that are able to describe itself. An early example of such a system is Meta II [54]. Object Grammars bring both notions of self-description together into the model of Figure 8.

Programming languages that are used to describe there own semantics are known at least since McCarthy's meta-circular Lisp 1.5 interpreter [43]. A programming language that is really defined in terms of its own concepts and structures leads to advanced forms of computational reflection [58] where the semantics of a programming language can be inspected and changed while it runs. In particular, this leads to a reflective tower of interpreters where the next level of semantics is defined in terms of layers below [30, 68]. A similar layering can be observed when changing any of the core models in Ensō. For instance, to change the grammar formalism, both the Grammar Grammar and the Grammar Schema must be changed to accomodate, for instance, a new syntactic construct. However the addition itself cannot be used yet in the grammar, since then the old grammar cannot be used to parse the new grammar.

Although the concept of bootstrapping is well-known, and widely practiced in the area of compiler construction, literature about the conceptual aspects of bootstrapping is surprisingly scarce. Appel provides an in-depth guide to bootstrapping in the context of ML [4]. Another in-depth discussion of how to bootstrap extensible object models is provided in [50, 51]. Bootstrap sequences can be described with T-diagrams, which are a visual formalism to reason about translator interactions [15]. These can be used to better understand complex bootstrapping processes given a number of language processors (compilers, interpreters, etc.) [39].

### *7.4. Discussion*

The requirements for mapping grammars to meta-models were first formulated in [33]: the mapping should be customizable, bidirectional and model-based. The Object Grammars presented in this paper satisfy these requirements. First, the mapping is customizable because of asynchronous binding: the resulting structures are to a large extent independent of the structure of the grammar. Path-based referencing and predicates are powerful tools to control the mapping, but admit a bidirectional semantics so that formatting of models back to text is possible. Formatting can be further guided using formatting hints. Finally, Object Grammars are clearly model-based: both grammars and schemas are themselves models, self-formatted and self-describing respectively. A comparative overview of systems to parse text into graph structures that conform to class-based meta-models can be found in [26].

To our knowledge, Object Grammars represent the first approach to mapping between grammars and meta-models that supports modular combination of languages. Xtext, EMFText, TCS, MontiCore, and TCSSL are implemented using ANTLR. ANTLR's LL(*) algorithm, however, makes true grammar composition impossible. Object Grammars, on the other hand, *are* compositional due to the use of the general GLL parsing algorithm [56]. Moreover, the use of a general parsing algorithm has the advantage that there is no restriction on context-free structure. For instance, the designer of a modeling language does not have to worry about whether she can use left-recursion or whether her grammar is within some restricted class of context-free grammars, such as LL($k$) or LR($k$). As a result, Object Grammars can be structured in a way that is beneficial for resulting structure, without being subservient to a specific parsing algorithm. Object Grammars share this freedom with other grammar formalisms based on general parsing, such as SDF [64] and Rascal [35].

The way references are resolved in Object Grammars bears resemblance to the way attributes are evaluated in attribute grammars (AGs) [48]. AGs represent a convenient formalism to specify semantic analyses, such as name analysis and type checking, by declaring equations between inherited attributes and synthesized attributes. The AG system schedules the evaluation of the attributes automatically. Modern AG systems, such as JastAdd [17] and Silver [63], support reference attributes: instead of simple values, such attributes may evaluate to pointers to AST nodes. They can be used, for instance, to super-impose a control-flow graph on the AST. Reference resolving in Object Grammars is similar to attributes: they are declarative statements of fact, and the system—in our case the *build* function—decides how to operationally make these statements true.

A similar approach is the Name-Binding Language (NaBL) [36]. NaBL supports the specification of name spaces, scoping rules and reference resolution strategies by attaching declarative, domain-specific attributes to abstract syntax patterns. The NaBL engine interprets such specifications to resolve name-based references, and records the result in a separate semantic index.

Object-grammars are different from both attribute grammars and NaBL, however, in that the focus is on graph-like object-oriented data models. Although some attribute grammar systems (e.g., Kiama [57]) can be used to evaluate attributes on arbitrary graphs, many systems take the AST or parse tree as the starting point. Moreover, path-based references only allow navigating the object graph without performing arbitrary computations, and without native support for name spaces and scoping. Extending the Ensō schema language with AG style attributes is an area for further research.

## 8. Concluding Remarks

### *8.1. Directions for Further Research*

Object Grammars represent a comprehensive approach to mapping textual syntax to object graphs. We have shown how this formalism can be used for the practical definition and composition of DSLs. Nevertheless, there are ample opportunities for further research. Below we briefly discuss two important directions.

*Expressiveness of Paths.* The first question to be addressed is: What is the expressiveness of path-based references? There seems to be a spectrum of approaches to reference resolution. Pure context-free grammars only admit tree structures and hence cannot be used to create cross links. On the other end, the traditional, separate specification of name analysis provides the most expressive power. Somewhere in the middle we find the automatic global reference resolution of Xtext [20], dedicated name resolution features of NaBL [36], and attribute grammar formalisms (e.g., [16]). Where to position Object Grammars on this spectrum? Intuitively it seems at least inbetween global reference resolution and NaBL. A direction of future work is thus to formally characterize what is possible with path-based references and what is not. This will involve experimenting with path-based encodings of name resolution concepts such as shadowing, name spaces, imports, overriding etc. We consider the features supported by NaBL to be a suitable benchmark.

*Static Guarantees.* What is the formal compatibility relation between an Object Grammar and its target schema? This question needs to be answered to be able to provide static guarantees with respect to success of parsing and formatting. In other words, is it possible to check that every string parsed by an Object Grammar produces an object graph conforming to the target schema? Conversely, how can we check that every object conforming to a schema, can be rendered to text using an object grammar? Similar questions can be asked in the context of Object Grammar composition. For instance: when is composition commutative? The current merge algorithm overwrites properties that are found in both models, so in general this does not hold. A related question is: how to statically check that if two Object Grammars $G_1$ and $G_2$ are compatible to schemas $S_1$ and $S_2$ respectively, the composition $G_1 \lhd G_2$ is compatible to $S_1 \lhd S_2$? Having automatic checks for such properties would make the use of Object Grammars less error-prone. Moreover, it would provide a stepping stone for advanced IDE support for Object Grammars, such as coupled refactoring of both grammar and schema [65].

We are currently exploring abstract interpretation as a way to infer the "implied" schema of an Object Grammar. This inferred schema could then form the basis for comparison with the intended schema. For parsing, the inferred schema should be subsumed by the intended schema: any object produced using the grammar should conform to the intended schema. For formatting, it is the other way round: any object conforming to the intended schema should also conform to the inferred schema. Techniques used in description logic could be used for computing such relations [6]. Note that the asynchronous nature of specifying object construction and field binding, the frequent occurrence of recursion in production rules, and the generality of predicates very much complicate such schema inference. Moreover, subsumption of schemas might not even be decidable in the general case.

### 8.2. Conclusion

Object Grammars are a formalism for bidirectional mapping between text and object graphs. Unlike traditional grammars, Object Grammars include a declarative specification of the semantic structure that results from parsing. The notation allows objects to be constructed and their fields to be bound. Paths specify cross-links in the resulting graph structure. Thus the result of parsing is a graph, not a tree. Object Grammars can also be used to format an object graph into text.

Our implementation of Object Grammars in Ensō supports arbitrary context-free grammars. This is required when composing multiple grammars together. We have presented elaborate details on how Object Grammars are realized in Ensō. Finally, we have shown how Object Grammars are used in Ensō to support modular definition and composition of DSLs.

*Acknowledgments.* We are thankful to the anonymous referees whose feedback lead to considerable improvement of this paper. We thank Ali Afroozeh for helping improve the parser.

## References

[1] M. Alanen, I. Porres, A Relation Between Context-Free Grammars and Meta Object Facility Metamodels, Technical Report 606, Turku Centre for Computer Science, 2004.

[2] S. Apel, D. Hutchins, A calculus for uniform feature composition, ACM Trans. Program. Lang. Syst. 32 (2008) 19:1–19:33.

[3] S. Apel, C. Kastner, C. Lengauer, FeatureHouse: Language-independent, automated software composition, in: Proceedings of the International Conference on Software Engineering (ICSE), pp. 221–231.

[4] A.W. Appel, Axiomatic bootstrapping: a guide for compiler hackers, ACM Trans. Program. Lang. Syst. 16 (1994) 1699–1718.

[5] P. Avgustinov, T. Ekman, J. Tibble, Modularity first: a case for mixing AOP and attribute grammars, in: Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), ACM, 2008, pp. 25–35.

[6] F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, P.F. Patel-Schneider (Eds.), The description logic handbook: theory, implementation, and applications, Cambridge University Press, 2003.

[7] K. Bąk, K. Czarnecki, A. Wąsowski, Feature and meta-models in Clafer: mixed, specialized, and coupled, in: Proceedings of the 3rd international conference on Software Language Engineering (SLE'10), Springer, 2011, pp. 102–122.

[8] J. Bézivin, On the unification power of models, Software and System Modeling 4 (2005) 171–188.

[9] M. Bravenboer, E. Visser, Parse table composition, in: Proceedings of the International Conference on Software Language Engineering (SLE). Revised selected papers., volume 5452 of *LNCS*, Springer, 2009, pp. 74–94.

[10] D.G. Cantor, On the ambiguity problem of backus systems, J. ACM 9 (1962) 477–479.

[11] A. Casey, L. Hendren, MetaLexer: a modular lexical specification language, in: Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), ACM, 2011, pp. 7–18.

[12] P.P. Chen, The Entity-Relationship Model—Toward a Unified View of Data, ACM Trans. Database Syst. 1 (1976).

[13] D. Crockford, The application/json Media Type for JavaScript Object Notation (JSON), RFC 4627 (Informational), 2006.

[14] DevBoost, EMFText: concrete syntax mapper, 2012. `http://www.emftext.org/`.

[15] J. Earley, H. Sturgis, A formalism for translator interactions, Commun. ACM 13 (1970) 607–617.

[16] T. Ekman, G. Hedin, Modular name analysis for Java using JastAdd, in: Proceedings of the International Summerschool on Generative and Transformational Techniques in Software Engineering (GTTSE), Springer, 2006, pp. 422–436.

[17] T. Ekman, G. Hedin, The JastAdd system—modular extensible compiler construction, Sci. Comput. Program. 69 (2007) 14–26.

[18] S. Erdweg, P.G. Giarrusso, T. Rendel, Language composition untangled, in: Proceedings of the International Workshop on Language Descriptions, Tools and Applications (LDTA).

[19] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W.R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P.J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, J. van der Woning, The State of the Art in Language Workbenches. Conclusions from the Language Workbench Challenge, in: M. Erwig, R.F. Paige, E.V. Wyk (Eds.), Proceedings of the Sixth International Conference on Software Language Engineering (SLE'13). In print.

[20] M. Eysholdt, H. Behrens, Xtext: implement your language faster than the quick and dirty way, in: OOPSLA Companion (SPLASH), ACM, 2010, pp. 307–309.
[21] D. Flanagan, Y. Matsumoto, The Ruby Programming Language, O'Reilly, 2008.
[22] F. Fondement, Concrete syntax definition for modeling languages, Ph.D. thesis, EPFL, 2007.
[23] F. Fondement, R. Schnekenburger, S. Gérard, P.A. Muller, Metamodel-Aware Textual Concrete Syntax Specification, Technical Report LGL-2006-005, EPFL, 2006.
[24] B. Ford, Parsing expression grammars: a recognition-based syntactic foundation, in: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'04), ACM, 2004, pp. 111–122.
[25] W. Frakes, C. Terry, Software reuse: metrics and models, ACM Comput. Surv. 28 (1996) 415–435.
[26] T. Goldschmidt, S. Becker, A. Uhl, Classification of concrete textual syntax mapping approaches, in: Proceedings of the European Conference on Model Driven Architecture—Foundations and Applications (ECMDA-FA), volume 5095 of *LNCS*, pp. 169–184.
[27] E. Goto, Monocopy and associative algorithms in an extended LISP, Technical Report 74-03, University of Tokyo, 1974.
[28] M. Hammer, D. McLeod, The semantic data model: a modelling mechanism for data base applications, in: Proceedings of the International Conference on Management of Data (SIGMOD), ACM, 1978, pp. 26–36.
[29] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, C. Wende, Derivation and refinement of textual syntax for models, in: Model Driven Architecture—Foundations and Applications (ECMDA-FA), volume 5562 of *LNCS*, Springer, 2009, pp. 114–129.
[30] S. Jefferson, D. Friedman, A simple reflective interpreter, LISP and Symbolic Computation 9 (1996) 181–202.
[31] S.C. Johnson, YACC—yet another compiler-compiler, Technical Report CS-32, AT & T Bell Laboratories, 1975.
[32] F. Jouault, J. Bézivin, KM3: A DSL for metamodel specification, in: Proceedings of the 8th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06), volume 4037 of *LNCS*, Springer, 2006, pp. 171–185.
[33] F. Jouault, J. Bézivin, On the specification of textual syntaxes for models, in: Eclipse Modeling Symposium, Eclipse Summit Europe 2006.
[34] F. Jouault, J. Bézivin, I. Kurtev, TCS: a DSL for the specification of textual concrete syntaxes in model engineering, in: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), ACM, 2006, pp. 249–254.
[35] P. Klint, T. van der Storm, J. Vinju, Rascal: A Domain Specific Language for Source Code Analysis and Manipulation, in: Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE, 2009, pp. 168–177.
[36] G. Konat, L. Kats, G. Wachsmuth, E. Visser, Declarative name binding and scope rules, in: K. Czarnecki, G. Hedin (Eds.), Software Language Engineering, volume 7745 of *LNCS*, Springer, 2013, pp. 311–331.
[37] H. Krahn, B. Rumpe, S. Völkel, Integrated definition of abstract and concrete syntax for textual languages, in: Proceedings of the International Conference On Model Driven Engineering Languages And Systems (MoDELS), volume 4735 of *LNCS*, Springer, 2007, pp. 286–300.
[38] I. Kurtev, J. Bézivin, F. Jouault, P. Valduriez, Model-based DSL frameworks, in: OOPSLA Companion (OOPSLA), ACM, 2006, pp. 602–616.
[39] O. Lecarme, M. Pellissier, M.C. Thomas, Computer-aided production of language implementation systems: A review and classification, Software–Practice and Experience 12 (1982) 785–824.
[40] A. Loh, Piping and instrumentation in Ensō, Language Workbench Challenge Workshop at Code Generation 2012, 2012. `http://www.languageworkbenches.net/index.php?title=LWC_2012`. Accessed March 23rd, 2013.
[41] A. Loh, T. van der Storm, W.R. Cook, Managed data: modular strategies for data abstraction, in: Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software (Onward! '12), ACM, 2012, pp. 179–194.
[42] L.T. Lukas Diekmann, Parsing composed grammars with language boxes, Workshop on Scalable Language Specification, 2013. Online: `http://tratt.net/laurie/research/pubs/papers/diekmann_tratt__parsing_composed_grammars_with_language_boxes.pdf`.
[43] J. McCarthy, LISP 1.5 Programmer's Manual, The MIT Press, 1962.
[44] B. Merkle, Textual modeling tools: overview and comparison of language workbenches, in: OOPSLA Companion (SPLASH), ACM, 2010, pp. 139–148.
[45] P.A. Muller, F. Fondement, F. Fleurey, M. Hassenforder, R. Schneckenburger, S. Gérard, J.M. Jézéquel, Model-driven analysis and synthesis of textual concrete syntax, Software and System Modeling 7 (2008) 423–441.
[46] Object Management Group, Unified Modeling Language Specification, version 1.3, OMG, `http://www.omg.org`, 2000.
[47] B.C. Oliveira, T.v.d. Storm, A. Loh, W.R. Cook, Feature-oriented programming with object algebras, in: ECOOP, volume 7920 of *LNCS*, Springer, 2013, pp. 27–51.
[48] J. Paakki, Attribute grammar paradigms—a high-level methodology in language implementation, ACM Comput.

Surv. 27 (1995) 196–255.
[49] T.J. Parr, R.W. Quong, ANTLR: a predicated-LL(k) parser generator, Softw. Pract. Exper. 25 (1995) 789–810.
[50] I. Piumarta, Accessible Language-Based Environments of Recursive Theories (a white paper advocating widespread unreasonable behavior), Technical Report RN-2006-001-a, Viewpoints Research Institute (VPRI), 2006.
[51] I. Piumarta, A. Warth, Open, extensible object models, in: R. Hirschfeld, K. Rose (Eds.), Self-Sustaining Systems, Springer-Verlag, 2008, pp. 1–30.
[52] T. Rendel, K. Ostermann, Invertible syntax descriptions: unifying parsing and pretty printing, in: Haskell'10, ACM, 2010, pp. 1–12.
[53] L. Renggli, M. Denker, O. Nierstrasz, Language boxes: bending the host language with modular language changes, in: Proceedings of the International Conference on Software Language Engineering (SLE), volume 5969 of *LNCS*, Springer, 2010, pp. 274–293.
[54] D.V. Schorre, META II a syntax-oriented compiler writing language, in: Proceedings of the 1964 19th ACM national conference, ACM'64, ACM, 1964, pp. 41.301–41.3011.
[55] A.C. Schwerdfeger, E.R. Van Wyk, Verifiable composition of deterministic grammars, in: Proceedings of the Conference on Programming Language Design and Implementation (PLDI'09), ACM, 2009, pp. 199–210.
[56] E. Scott, A. Johnstone, GLL parse-tree generation, Science of Computer Programming 78 (2013) 1828–1844.
[57] A.M. Sloane, L.C. Kats, E. Visser, A pure embedding of attribute grammars, Science of Computer Programming 78 (2013) 1752–1769.
[58] B.C. Smith, Reflection and semantics in LISP, in: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL'84), ACM, 1984, pp. 23–35.
[59] T. van der Storm, W.R. Cook, A. Loh, Object grammars: Compositional & bidirectional mapping between text and graphs, in: Proceedinngs of the 5th International Conference on Software Language Engineering (SLE'12), volume 7745 of *LNCS*, Springer, 2012, pp. 4–23.
[60] M. Tomita, LR parsers for natural languages, in: Proceedings of the 10th international conference on Computational linguistics (COLING'84), pp. 354–357.
[61] L. Tratt, Direct left-recursive parsing expression grammars, Technical Report EIS-10-01, School of Engineering and Information Sciences, Middlesex University, 2010.
[62] E. Van Wyk, Aspects as modular language extensions, in: Proc. of Language Descriptions, Tools and Applications (LDTA), volume 82.3 of *Electronic Notes in Theoretical Computer Science*, Elsevier Science, 2003.
[63] E. Van Wyk, D. Bodin, J. Gao, L. Krishnan, Silver: an extensible attribute grammar system, Science of Computer Programming 75 (2010) 39–54.
[64] E. Visser, Syntax Definition for Language Prototyping, Ph.D. thesis, University of Amsterdam, 1997.
[65] J. Visser, Coupled transformation of schemas, documents, queries, and constraints, Electronic Notes in Theoretical Computer Science 200 (2008) 3–23.
[66] M. Völter, Language and IDE modularization and composition with MPS, in: Generative and Transformational Techniques in Software Engineering IV (GTTSE'11). Revised Papers, volume 7680 of *LNCS*, Springer, 2013, pp. 383–430.
[67] M. Völter, E. Visser, Language extension and composition with language workbenches, in: OOPSLA Companion (SPLASH), ACM, 2010, pp. 301–304.
[68] M. Wand, D. Friedman, The mystery of the tower revealed: A nonreflective description of the reflective tower, LISP and Symbolic Computation 1 (1988) 11–38.
[69] A. Warth, J.R. Douglass, T. Millstein, Packrat parsers can support left recursion, in: Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation (PEPM'08), ACM, 2008, pp. 103–110.
[70] Wikipedia, Piping and instrumentation diagram—Wikipedia, The Free Encyclopedia, 2012. `http://en.wikipedia.org/w/index.php?title=Piping_and_instrumentation_diagram`. Accessed March 23rd, 2013.
[71] M. Wimmer, G. Kramler, Bridging grammarware and modelware, in: Proceedings of the Satellite Events at the MoDELS Conference, Springer, 2006, pp. 159–168.
[72] N. Wirth, What can we do about the unnecessary diversity of notation for syntactic definitions?, Commun. ACM 20 (1977) 822–823.