Cedalion: A Language for Language Oriented Programming *

David H. Lorenz Boaz Rosenan

Open University of Israel,
1 University Rd., P.O.Box 808, Raanana 43107 Israel
lorenz@openu.ac.il brosenan@cslab.openu.ac.il

Abstract

Language Oriented Programming (LOP) is a paradigm that puts domain specific programming languages (DSLs) at the center of the software development process. Currently, there are three main approaches to LOP: (1) the use of *internal DSLs*, implemented as libraries in a given host language; (2) the use of *external DSLs*, implemented as interpreters or compilers in an external language; and (3) the use of *language workbenches*, which are integrated development environments (IDEs) for defining and using external DSLs. In this paper, we contribute: (4) a novel language-oriented approach to LOP for defining and using internal DSLs. While language workbenches adapt internal DSL features to overcome some of the limitations of external DSLs, our approach adapts language workbench features to overcome some of the limitations of internal DSLs. We introduce Cedalion, an LOP host language for internal DSLs, featuring static validation and projectional editing. To validate our approach we present a case study in which Cedalion was used by biologists in designing a DNA microarray for molecular Biology research.

Categories and Subject Descriptors D.1.6 [*Programming Techniques*]: Logic Programming—DSLs; D.2.6 [*Software Engineering*]: Programming Environments—Programmer workbench; D.3.2 [*Programming Languages*]: Language Classifications—Extensible languages.

General Terms Design, Languages.

Keywords Language-oriented programming (LOP), Language workbenches, Logic programming, Domain-specific languages (DSL).

* This research was supported in part by the *Israel Science Foundation (ISF)* under grant No. 926/08.

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.

1. Introduction

Language Oriented Programming (*LOP)* [5, 7, 34][1] is a paradigm that puts domain specific programming languages (DSLs) at the center of the software development process. LOP is known by different code names. At Microsoft, LOP was named *Intentional Programming* [27]. That project lead to the formation of the *Intentional Software* company, which uses that name to describe their version of LOP [28]. Other terms related to LOP include *Concept Programming* in the XL programming language [36] and *Dialecting* in REBOL [25]. *Model Driven (Software) Development (MDD/MDSD)* [19, 30] is also a related term, when considering domain-specific modeling languages as DSLs [10].

The LOP software development process consists of three steps: (1) a definition of a DSL or several interoperable DSLs; (2) the implementation of these DSLs, e.g., by means of interpreters, translators or compilers; and (3) the development of the software using these interoperable DSLs. LOP software development is often viewed as working "*from the middle out*." The DSL definitions (Step 1) come first, followed by development of the application on top (Step 3), which can be done in parallel with the underlying implementation of the DSLs (Step 2).

1.1 Classification of LOP Environments

We use the term *LOP environment* to generally refer to a language or a tool facilitating LOP software development. There are currently three general categories of LOP environments [7], each representing a unique approach to LOP. We discuss each of the categories briefly here. Table 1 provides a summary of their trade-offs.

(I) *Internal DSLs* (or embedded [13] DSLs) are DSLs implemented from within a host programming language. Code in an internal DSL is actually code in the host language. These DSLs are easy and thus cost-effective to implement, and enjoy interoperability "out of the box." However, the syntax and semantics of the DSL are subject to the constraints of the host language. Tools provided for

[1] To the best of our knowledge, the term LOP was coined by Ward [34] and adopted by Dmitriev [5] and Fowler [7].

DSL Development	(I) Internal DSLs	(II) External DSLs	(III) Lang. Workbench	(IV) CEDALION
Freedom in definition	○	●	●	●
Cost-effectiveness	use—◑—implementation	○	use—◐—implementation	●
Interoperability	●	○	●	●

Table 1: Properties of LOP environments

the host language can be used when working with internal DSLs, but since these tools are not made for DSLs, productivity when using them does not match that of custom DSL tooling.

(II) *External DSLs* are DSLs implemented as compilers or interpreters, and are thus external to the language in which they were implemented. Because they are implemented externally, their developers enjoy more "freedom of expression" in defining them, and are only bound by the limitations of the tools they use, typically compiler-generation tool chains, such as Lex and Yacc (which they are free to choose). However, that freedom comes at a price. Implementing external DSLs is usually costly. Traditional tool chains do not support DSL interoperability, and leave the construction of productivity tools to the DSL developer, thereby making the use of these DSLs also cost-ineffective.

(III) *Language workbenches* [5, 7, 14, 28] are integrated development environments (IDEs) for defining, implementing and using external DSLs. They combine the features of external and internal DSLs to provide a better solution for LOP. They use external DSLs as a stepping stone and add a common representation for all DSLs, a feature associated with internal DSLs. Working with external DSLs, language workbenches retain their freedom in defining DSLs, but also support interoperability between DSLs. They provide tooling to make the use of DSLs more practical. They also provide their own meta-DSLs to facilitate DSL implementation, but this part is still not as cost-effective as it is for internal DSLs [17].

1.2 Contribution

This paper introduces Cedalion—an LOP environment of a new sort, and a category in itself, we name *LOP languages:*

(IV) *LOP languages* present a novel approach to LOP (Figure 1) . Instead of taking external DSLs and giving them internal DSL tooling (Figure 1, III), LOP languages take internal DSLs and give them external DSL tooling (Figure 1, IV). Cedalion is an example of such an LOP language. It is the first LOP environment to feature hosting of internal DSLs, projectional editing [8], and static validation. Collectively, these help to overcome the inherent restrictions on syntax and semantics definition associated with the use of internal DSLs.

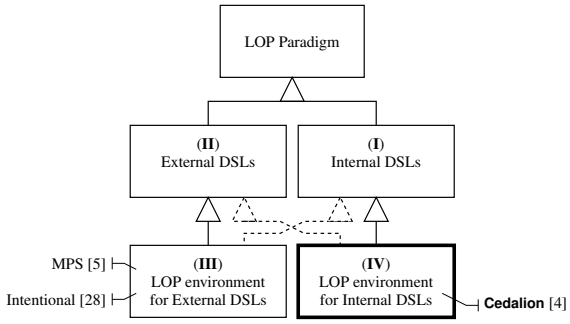


Figure 1: Classification of LOP environments

The main observation is that internal DSLs can provide a much better stepping stone for an LOP environment than external DSLs. Internal DSLs are naturally interoperable and are already cost-effective with respect to DSL implementation. Their main limitation is the constraints imposed by the hosting language on the DSL definition. However, this limitation seems to be easier to bridge, relative to the limitations posed by external DSLs (Table 1).

Since we use internal DSLs as the stepping stone, our LOP environment solution is essentially a host language. We call such a host language for LOP a *workbench language* (as opposed to *language workbench*) or, to avoid confusion, simply: an *LOP language*.

LOP languages contest language workbenches as the sole practical approach to LOP. We demonstrate that an LOP language can provide a viable alternative to a language workbench in combining the advantages of internal DSLs with the flexibility associated with external DSLs. Specifically, we contribute a concrete implementation of an LOP language: Cedalion [4, 16, 23].

1.3 Evaluation

As evidence of the effectiveness of Cedalion as an LOP language and environment, we present for illustration a small time schedule example (Section 4) and for validation a real-life Bioinformatics case study (Section 5). The effectiveness is further evident from the use of Cedalion in the implementation of parts of Cedalion itself and its DSL libraries. These examples demonstrate the viability and versatility of an LOP language as an LOP environment.

2. Properties of LOP Environments

In order to reason about LOP environments, we identify the following indispensable yet incomplete aspects of nearly every LOP environment ("*DSL Bill of Rights*"):

- *Freedom of expression (DSL definition*): An ideal LOP environment poses almost no syntactic or semantic limitations on defining DSLs, allowing DSL code to be expressed in the way that is most familiar and intuitive to domain experts.
- *Economic freedom (DSL implementation and use):* An important goal of an LOP environment is to make LOP practical and cost-effective. One of the prohibitive factors in the practicality of LOP is the cost of implementing the DSLs (the cost of going down the LOP process). A good LOP environment reduces this cost enough to make LOP cost-effective. However, implementing the DSL is not enough. Using it (the cost of going up the LOP process) must be cost-effective as well. The level of tooling, such as syntax highlighting and auto-completion, determines the effectiveness of the LOP environment.
- *DSLs' freedom of association (DSL interoperability*): The true benefit of LOP is realized when the LOP environment enables separate, reusable DSLs to be used together to implement a complex system [17]. In particular, interoperability means that phrases in one DSL can be embedded inside phrases in another DSL, to express multi-paradigm behavior [31].

Next, we elaborate on each of these aspects.

2.1 Freedom of Expression (DSL Definition)

Freedom in DSL definition is the most basic characteristic of an LOP environment. In LOP, we need to be able to define DSLs to match the problem at hand. To be able to do that, we have to be able to define a DSL exhibiting the *syntax* and the *semantics* desired by the domain expert.

Syntactic Freedom A good LOP environment needs to give its users the freedom to define DSLs that resemble, as much as possible, the notation used by the domain experts. For example, consider a DSL for defining sets, with a language construct for the *union* operation. The natural mathematical notation from set theory (for the union of sets A and B) is:

$$A \cup B$$

Ideally, we should be able to make the syntax for that construct in our DSL identical to its mathematical representation.

One problem here would be the fact that $\cup$ is not an ASCII character, and thus unacceptable in ASCII-based languages. An acceptable solution for that might be to use instead:

$$A \,\mathsf{union}\, B$$

However, using pre-fix syntax such as

$$\mathsf{union}(A, B)$$

or

$$(\mathsf{union}\ A\ B)$$

would be unadvised, since large expressions may be hard to read that way.

In general, some restrictions on syntax such as conformance to a grammar class, e.g., LALR(1) or LL(k), are acceptable here, but may not be acceptable when DSL interoperability is considered (Section 2.3). Other restrictions, such as confinement to ASCII characters or a certain lexical structure, can be considered, but obviously, the less restrictions the better. For Cedalion, our requirement is to enable the more natural mathematical notation in this case (see Section 3.4).

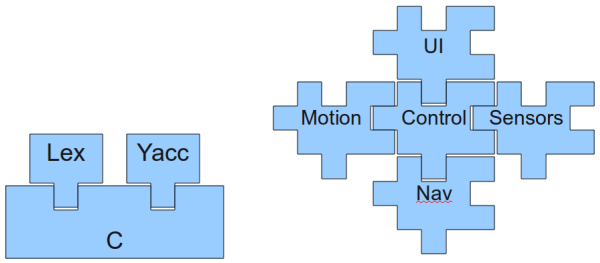
Semantic Freedom Semantic freedom applies to both static and dynamic semantics. An important requirement from an LOP environment is the ability to statically validate that the DSL code is well-formed, so that errors can be detected early, and in some cases, tooling may help write valid code to begin with. An LOP environment should give DSL designers the freedom to define any *static semantics* rules of validity that make sense for their own DSL.

DSLs created on an LOP environment should also be able to have any *dynamic semantics*. For example, *fluent interfaces* [6], sometime used as textual DSLs (e.g., jMock call-chains [9]) or visual DSLs (e.g., JavaBeans event-graphs [18]), are very limited in their *meaning*. They are only capable of describing things that can be represented by chained method calls.

Many host languages for internal DSLs are very good at providing freedom in defining dynamic semantics, but because of their dynamic nature, they often do not support custom static semantic rules for DSLs. For Cedalion, we require both static and dynamic semantic freedom to be supported.

2.2 Economic Freedom (Cost-effective DSL Implementation and Use)

Cost-effectiveness is necessary for making LOP a viable approach in software development. All too often, software developers dismiss DSLs and use general-purpose languages instead, due to the perceived cost of developing them. In addition to the DSL development effort, one must take into account the cost of using DSLs. DSLs are made to be more concise than code in a general-purpose language, but general-purpose languages often come with rich tools such as smart editors, debuggers, profilers, etc., that compensate for the verbosity of the language. When a DSL is lacking this kind of tools, productivity is harmed. However, providing such tools for each DSL can be prohibitively costly too [20].



(a) Interoperability through a general-purpose language

(b) Interoperability of DSLs

Figure 2: DSL interoperability illustrated

For Cedalion our requirement here is therefore twofold. On the one hand, we require a cost-effective method for DSL implementation. On the other hand, we require that the cost of providing reasonable tooling for these DSLs will be low. Only a fulfillment of both requirements will enable a cost-effective LOP process.

2.3 DSLs' Freedom of Association (DSL Interoperability)

DSLs provide good tools for solving problems in their own domains, but practical software systems are never about a single problem domain. In fact, most real-life software systems work with multiple domains. One can claim that the software of an auto-pilot mechanism of a light aircraft is "all about" nested control loops. However, what good will that software do if it cannot communicate with the motion drivers steering the plane, and the sensors reading flight information? Also, while piloting in a straight line is good enough for limited purposes, a good auto-pilot system should be able to connect to the plane's navigational system and make geodetic computations to follow a flight plan. All these are problem domains that require their own DSLs.

DSLs' freedom of association is what we call the ability to have several DSLs work together to build large, complex, multi-paradigm software, such as the software for an auto-pilot system. Traditionally, DSL interoperability has been achieved by having each DSL interact with a general-purpose programming language. An example, depicted in Figure 2a, is the cooperation of Lex and Yacc (and their alike), by integration to C. Both Lex and Yacc use actions written in C, and the overall scanner/parser is translated into C code. The C code produced by Lex and Yacc is then combined to create a complete scanner-parser.

With LOP we want to avoid the need for "glue code." We want the DSLs to interact with one another directly. Just like Yacc files contain embedded C code, we wish to embed code in one DSL inside code in another DSL. In the auto-pilot example this means, e.g., that motion code can be embedded inside control loops (Figure 2b). It is obvious that this kind of interoperability requires integration on both the *syntactic* and the *semantic* levels.

Syntactic Interoperability This requires that the syntax of multiple DSLs can be combined into a syntax of a combined language. This requirement is far from trivial. For example, if our LOP environment uses a parser generator that requires grammars to be LALR(1), it is not guaranteed that combining two valid LALR(1) grammars will emit a valid LALR(1) grammar. Moreover, when combining two unambiguous grammars, there is no guarantee that the resulting grammar will remain unambiguous. Syntactic interoperability requires a method of dealing with such ambiguities.

Semantic Interoperability When combining DSLs, their implementations need to interact, in order to provide a unified semantics. This requires some notion of common representation for all DSLs provided by the LOP environment. In addition, it requires coordination between the DSL implementations.

3. Cedalion in a Nutshell

Cedalion is a logic programming, LOP language and an IDE workbench for hosting internal DSLs. Cedalion represent a novel approach to LOP, one that like language workbenches, bridges the gap between internal and external DSLs, allowing its users to enjoy the best of both worlds. However, unlike language workbenches, which start with external DSLs and add a feature of internal DSLs (namely, a common representation for DSLs), Cedalion starts with internal DSLs, and adds language workbenches features (namely, projectional editing and static validation) to the host programming language.

However, interestingly enough, while projectional editing is used in language workbenches to support syntactic interoperability for external DSLs, it is used in Cedalion to enable syntactic freedom in defining internal DSLs. The other feature we take from language workbenches, static validation capabilities, is used in Cedalion to check DSL code as the code is being edited, and for providing tooling support (e.g., auto-completion) based on that.

3.1 The Cedalion Workbench

Cedalion code cannot be edited using traditional text editors and can only be edited using a special *projectional editor*, a dedicated editor for projectional editing. The Cedalion *workbench* [4], an Eclipse-based IDE, implements such a projectional editor. *Projectional editing* offers an alternative to the traditional parsing approach. With projectional editing, instead of editing the code in a text editor and then parsing it to form an abstract syntax tree (AST) of the code, we edit the AST directly, and present it to the user using a *projection*, that is, a transformation to some human readable representation, which is usually (but not necessarily) textual.

Figure 3 is a screenshot of the Cedalion workbench. The editor screen is structured similar to the user interface of

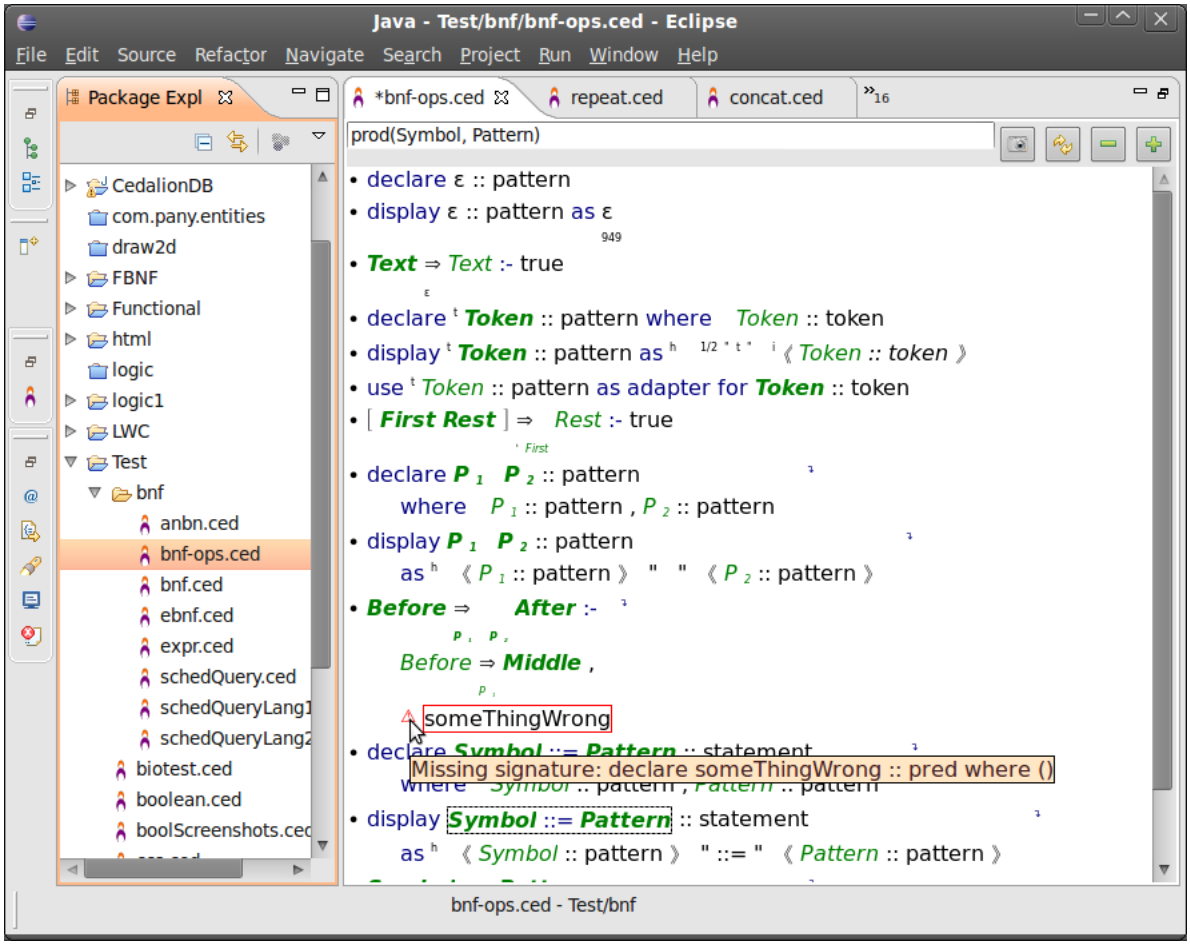


Figure 3: A screenshot of the Cedalion workbench

a Web browser. The top of the window contains a *text bar* (similar to the address bar of a Web browser) with a few action buttons to its left. The text bar displays the text representation of the currently selected Cedalion code. The rest of the window's real-estate (*code area*) is dedicated to projecting the content of the file being edited. The tab label indicates the name of the file.

Even at first glance, one can see that the Cedalion workbench differs from a text editor. The projected Cedalion code contains special symbols, and it is displayed using varying font sizes and unorthodox layout. We will explain the meaning of this code in Section 4. The code comprises a hierarchy of rectangular elements (we call *terms*), nested inside one another. When the user clicks on a term, a selection box appears around it, and the content of the text bar is replaced with a textual "Prolog-like" projection of that term. This textual representation can be edited, and when hitting Enter, the changes are applied to the code area, assuming the text complies with the simple Prolog-like syntax.

The edited code can be either valid or invalid Cedalion code. Unlike many syntax driven editors, Cedalion does allow invalid code to be edited. For example, an undefined term can be used. In such a case, Cedalion will mark this term with an error marker (a red rectangle with a small red warning sign symbol at its top left), and provide the details of the error as a tooltip. As can be seen in Figure 3, in the case of an undefined concept, Cedalion claims a missing signature, and uses type inference to suggest what this signature might be. Indeed, double clicking the red warning sign will suggest inserting such a type signature before the current statement. This is the way new concepts can be introduced. Concepts and type signatures are discussed in Sections 3.2 and 3.5, respectively.

3.2 Abstract Syntax

Figure 4 shows an entity diagram of the structure of a Cedalion program. A *program* consists of multiple *statements*. A statement consists of a single *compound term*, and provides scope for *logic variables*. A compound term holds an identifier, associating it with a *concept* that determine its meaning and its *arity* (the number of arguments it takes). According to its arity, each compound term takes zero or more arguments, which are themselves *terms*. A term can be ei-

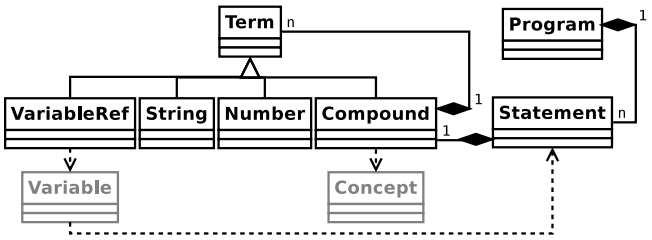


Figure 4: Entity diagram of the Cedalion program structure

ther a *number*, a *string*,[2] a *variable reference*, or a compound term. The latter allows compound terms to be complex.

Concepts and variables are depicted in Figure 4 in gray, since they are not an actual part of the AST structure, but rather are defined implicitly by referencing them from compound terms and variable references, respectively. Reference to both is done through an identifier. Identifiers used in variable references to specify a variable are scoped within the enclosing statement. The identifiers used in compound terms to specify a concept are globally scoped. To avoid name collisions, all concept identifiers in Cedalion contain a *namespace prefix*. This is a string that represents the package in which this concept has been defined, or sometimes the DSL to which the concept belongs. Informally, *DSLs* can be seen as sets of concepts.

Statements form directed acyclic graphs (DAGs). A compound term forms a tree, where each internal node is a compound term with a non-zero arity, and the leafs are compound terms with zero arity (atoms), numbers, strings, and variable references. Linking the variable references to the (implicit) variables they refer to, forms a DAG, since a variable can be referenced more then once in the same statement.

3.3 Dynamic Semantics

The dynamic semantics of Cedalion is based on that of Prolog, with logic variables and predicates playing a major role in its semantics.

Logic Variables Cedalion's *logic variables* are born unbound, and can be bound to a term. Once bound to a term, they do not change their value, unless the interpreter backtracks beyond the point in time when they were bound. Cedalion has the same notion of *term unification* as Prolog. This is a process in which two terms are being compared to see if they *match* (i.e., there exists an assignment to their variables that would make them equal).

Predicates *Predicates* are concepts that define a relation. Their instances (compound terms for which their concept is a predicate) are called *goals*. A goal can have zero or more *solutions*, i.e., assignments to the variables referenced from within the goal for which the resulting terms satisfy the relationship defined by the predicate.

[2] Unlike Prolog, where atoms (compound terms with zero arguments) are used as strings

Like in Prolog, running a Cedalion program is done by querying a goal. The result is given in the form of zero or more variable assignments. However, unlike Prolog, Cedalion predicates do not have side-effects, so Cedalion predicates are limited to pure logic operations.

Cedalion provides around forty *built-in predicates*, which are predicates that provide functionality otherwise not available to Cedalion programs. None of Cedalion's built-in predicates have side effects. Cedalion programs extend this pool of predicates by defining new predicates. Similar to Prolog, this is done using *Horn clauses*, which are statements of the form:

Head :- *Body* (Clause)

where both *Head* and *Body* are goals. However, different than Prolog, in Cedalion this is merely a textual representation for an abstract structure, where a Horn Clause statement has a compound term of the concept named :-, with two arguments, named *Head* and *Body*. Generally, this clause means that a goal matching *Head* is true for every solution of *Body*.

Each predicate can have one or more Horn clauses defining its semantics. As in Prolog, they are joined through disjunction, so a goal has a solution if it satisfies *any* of the clauses defining the predicate it is based on. Note that clauses in Prolog have either the form *Head* :- *Body*. or the form *Head*. without a body, which is equivalent to *Head* :- true.. Cedalion does not support clauses without bodies, so the form *Head* :- true. should be used in such cases.

Rewrite Rules Cedalion supports user defined statements. These are statements that are not Horn clauses themselves, but can be translated to Horn clauses using *rewrite rules*. Rewrite rules are statements of the form:

$$S_1 \rightsquigarrow S_2$$

which gives S_1 meaning in terms of S_2, that is, for every statement S_1' matching S_1, a statement S_2', which is given by S_2 under the same variable assignment, is contributed. This rewrite rule will define semantics for statements matching S_1 when either S_2 is a Horn clause, or rewrite rules exist such that S_2 can be expressed in terms of a Horn clause. Rewrite rules are not destructive. Applying $S_1 \rightsquigarrow S_2$ does not remove S_1, and it can still be used in other rewrite rules.

3.4 Projection Definition

So far we discussed Cedalion's abstract syntax and dynamic semantics. We did not (yet) discuss concrete syntax, since this is only provided by Cedalion's projectional editing, which is based on its dynamic semantics.

Cedalion uses projectional editing to allow DSL developers to freely define the syntax of the DSL, regardless of the limitations of any parsing algorithm, character set (e.g., ASCII), or even reading order (e.g., left to right, top to bottom). DSL syntax can include variable font

sizes, styles and colors. A DSL may include special symbols and even some interaction capabilities, such as collapse/expand capabilities. Some of these capabilities (font style and color, collapse/expand) are available as text editor features within IDEs for some programming languages. However, in Cedalion, these features are part of the DSL's *concrete syntax*.

The concrete syntax of DSL concepts is determined by their *projection definitions*. A projection definition is a statement that define how a certain concept is displayed. This is similar to a production rule in a grammar, only that instead of parsing a string, here we render a visualization. A projection definition has the following form:

display *Term* :: *Type* **as** *Projection* (Projection definition)

where *Term* is a prototype of the concept we would like to refer to (i.e., a compound term that belongs to that concept, where all its arguments are variable references), *Type* is its type (to be discussed in Section 3.5), and *Projection* is a term describing how *Term* is to be visualized. *Projection* can use any of Cedalion's visualization primitives, providing labels, symbols, layouts, styles and interactions. Cedalion provides around twenty such primitives, and more can be added by extending the Eclipse plug-in (each primitive is implemented as a Java class, associated to the Cedalion code).

For example, the projection definition of a Horn clause is:

display (*Head*:-*Body*) :: statement
as ^{h} ⟨⟨*Head* :: pred⟩⟩ ":-" ⟨⟨*Body* :: pred⟩⟩

Here we use three kinds of visualization primitives: (1) a horizontal layout, represented by a whitespace-separated list, annotated with a tiny "h" to its top-left; (2) a label, represented by a double quoted string; and (3) place-holders for arguments, represented by double angle brackets, containing each argument along with its type. Note how the concept *Head*:-*Body* is depicted using its projection, even in its own projection definition.

Projection definitions are not mandatory. Cedalion defines Prolog-like default projections for concepts. A compound term with no arguments is projected as a label containing its identifier, without namespace prefixing (e.g., a compound term of concept a is projected as "a"). A compound term with arguments is projected as a label with its identifier, followed by parentheses, containing a comma-separated list of the arguments (e.g., concept a with arguments 1 and 2 is projected as "$a(1, 2)$"). Projection definitions are desired wherever the default rendering is unsatisfactory. For example, the union operator discussed in Section 2.1 can be defined without a projection definition, but then it would appear in the form union(A, B). A projection definition can do better than that, displaying it as $A \cup B$.

Cedalion's projectional editing is implemented in Cedalion. The Cedalion workbench queries a predicate defined by the Cedalion program to know how to visualize a certain term. This predicate calls an internal predicate that provides user-defined projections. Projection definitions contribute results to that predicate, using a rewrite rule.

In addition to providing Cedalion's concrete syntax, the projection definition mechanism provides hooks for Cedalion code to provide *checkers* (predicate clauses that examine the code that is being edited), and provide *markers* (annotations on the code). Markers can, for example, report errors associated with a certain term. This mechanism opens the door for users to define static semantics for their DSLs, and for Cedalion to define its *type system*.

3.5 Cedalion's Type System

Cedalion is statically typed. Its type system is completely implemented in Cedalion and runs as a set of checkers, from within the projectional editor. As such, it can be extended by Cedalion code.

Like most type systems for typed logic programming [26, 29], Cedalion's type system is implicit with regard to logic variables and explicit with regards to concepts. It infers the type of logic variables, but requires that every concept being used shall have a *type signature* (stating the concept's type and the types of its arguments), although, in most cases, that type signature can be inferred during editing. Type errors are presented too during editing, well before execution/interpretation.

Unlike the traditional, conservative approach [22], taken by most typed logic programming languages [26, 29], Cedalion allows some of the type checking to be performed at runtime. In a case where not all types are known during editing, e.g., some types are parametric and conveyed by variables, Cedalion's type system insists that these variables are to be exposed to the runtime environment. In such cases, the runtime environment will ensure type safety, knowing the actual types being used.

Typed Terms One important example of this is the *typed term*. A typed term is a pair of the form:

Term :: *Type* (Typed term)

where *Term* is a term of type *Type*. This is a type safe way to pass arguments regardless of their type, making their type available for checking at runtime. In Section 3.4 we already saw examples for typed terms in projection definitions.

Type Signatures Another example of where typed terms are in use is *type signatures*. A type signature is a statement that declares a new concept, by assigning types to itself and its arguments. A type signature has the form:

declare *Concept* **where** *ArgList* (Signature)

where *Concept* is a typed term, containing the concept being defined along with its type, and *ArgList* is a list of typed

terms, providing the types of the arguments. For example, the type signature of a typed term is:

```
declare (Term :: Type) :: typedTerm
  where Term :: Type, Type :: type
```

Note the difference between *Type* (a type variable, containing the *Term*'s type), and "type"—a concrete type, the type of all Cedalion types. Another interesting example is the type signature of the type signature statement. Since Cedalion's type system is implemented in Cedalion, its type signature statement has a type signature. It looks like this:

```
declare (declare Concept where ArgList) :: statement
where Concept :: typedTerm, ArgList :: list (typedTerm)
```

3.6 Cedalion's Tooling Support

The fact that Cedalion code is edited using a projectional editor, allows productivity features to be integrated into the language. Here we discuss some of them.

Context Menu Cedalion provides a context menu for every term. Right clicking a term will cause a pop-up menu to appear, listing operations relevant to this term. Selecting this operation will execute it, performing an action such as modifying code or displaying content in the *view area* (a part of the Cedalion workbench made for displaying information and interacting with the user). User code can contribute context menu entries by using *context menu entry* statements. These statements associate the caption of the entry with an action. They also specify a pattern for the term to be matched, in a form of a typed term. This allows entries to be specified for specific concepts, or specific types, thus making the menu context-dependent.

Auto-completion Auto-completion is the feature that is used the most when editing Cedalion code. When entering text in the text bar, the user can press a key combination (Control+Space) to get a list of suggestions. Cedalion queries the collection of concepts that can be used in the selected location, and finds ones which have *aliases* (explained next) starting with the string entered. The list is then presented to the user for choosing the appropriate concept, which is then inserted as a new term.

Aliases Aliases are strings associated with concepts. Each concept has a "natural" alias, which is its internal identifier (the name Cedalion uses internally, regardless of projection), without namespace prefixing. Additional aliases can be defined by the user. Cedalion also infers aliases in some cases from projection definitions (e.g., when the projection is a label, the content of the label is used as an alias for this concept). Concepts and aliases have a many-to-many relationship, where a single concept can have multiple aliases (e.g., its internal name and something based on its projection), and several concepts can share the same alias. In the latter case, disambiguation is done by choosing the desired entry from the auto-completion list of choices.

Adapters An adapter of type T_1 to type T_2 is a concept of type T_2, which takes one argument of type T_1, and semantically acts as a proxy, adding no additional meaning to its argument. Cedalion has a special declaration for declaring a concept as an adapter. This allows Cedalion to reconcile concepts of type T_1 in the context where a concept of type T_2 is needed. Adapters allow Cedalion's auto-completion to offer concepts of type T_1 in these cases, and when a type mismatch between T_1 and T_2 is presented, Cedalion automatically inserts the adapter to fix this error.

Definition Search Concepts are centric to the way Cedalion code is programmed. Concepts are introduced in Cedalion in both the DSL definition and the DSL code. To allow Cedalion users to be able to understand the different concepts and be able to track their definitions, Cedalion provides a mechanism for searching concept definitions. When selecting a compound term, Cedalion's context menu displays the option "Show Definitions." Selecting this option will display the full story behind the concept associated with the selected term. This "story" includes all aliases assigned to this concept, the type signature, projection definition and all semantic definitions. The nature of the semantic definitions for a concept depend on the concept type. For example, for predicates, the semantic definition includes all clauses contributing results to that predicate. A semantic definition of a statement includes all rewrite rules translating this statement into others. For a type, it includes all type signatures of concepts of that type. The user can relate new defining statements to concepts using "defines" statements. Each defining statement is displayed along with the file name in which it is defined. Clicking that definition will open that file, and highlight the relevant definitions with a green background.

3.7 Implementation

Cedalion is implemented as an open source project [4], in a pre-alpha maturity state. It is implemented as an Eclipse plug-in, in Java (a little over 5K lines of code), Prolog (less than 500 lines of code) and Cedalion (about 700 statements).

The Java code provides integration to Eclipse and to the Prolog engine (SWI Prolog [35]), and provides visualization and interaction capabilities, such as the Cedalion editor window, a collection of figures to display, context menu capabilities, etc. The Java code does not have any knowledge of the Cedalion programming language nor of its DSLs. It merely provides the "physical" capabilities.

The Prolog code implements a *logic engine*, i.e., a software module capable of storing facts and deduction rules, and answering queries, supporting namespaces and rewrite rules. To communicate with the part of Cedalion implemented in Java, the Prolog part implements the server side of that communication protocol. It also implements Cedalion's built-in predicates.

The Cedalion code implements Cedalion's advanced features, such as its projection editing, type system, and tooling

support, in what we call the *bootstrap package*. These features are implemented in a pure logic fashion, as answers to queries. The queries originate from the Java code through a set of predicate we call the Cedalion *Public Interface* (*CPI*). These predicates define the contract between the Java code (the *front-end*) and the Cedalion code (the *back-end*). For example, CPI contains a predicate for querying what needs to be displayed in a certain segment of a source file. The bootstrap package implements this predicate to consult the projection definition of the term in that segment to provide an answer. The concepts in the answer correspond to Java classes implementing the actual visuals to be displayed. The formulation of the CPI allows the front-end to be replaced by a different implementation (e.g., one not based on Eclipse), so long as it implements its side of the CPI.

4. Example: LOP with Cedalion

To demonstrate LOP with Cedalion we present in this section a complete step-by-step example. The scope of the example was deliberately narrowed down to allow us to provide the complete source code, allowing the reader to extrapolate this example to larger problems.

Consider the problem of providing free text queries in websites of transportation carriers, such as train, bus, and air operators. An example of a query may be to find the schedule for trains "from Paris to Rome today." The assumption is that these queries, while being "free text," are actually confined to a finite (and relatively small) language, since they all have the same elements: source, destination and time. In this example, we shall build a parser for this language. Our input is a list of tokens, provided by a smart lexical analyzer (or scanner), which, for the simplicity of our example, we assume is smart enough to understand date patterns and names of places. For starters, our output will be Boolean, indicating whether or not the list of tokens is a legal query. Later in this example we will extend our code to provide more informative output.

About Code Samples in this Paper Because Cedalion uses projectional editing, displaying Cedalion code should be done through its projection. For this reason we use the Cedalion workbench's built-in code snippet mechanism, which allows us to save Cedalion code fragments into an image files. For the purpose of this paper, we use numbered lists, so that statements can be referred to from the text. We use the numbering *Statement* $n.m$ to reference the statement at line m in *Figure* n.

4.1 Step 1: DSL Definition

Following the LOP approach, we do not resort to an ad-hoc solution to this problem (i.e., implement a parsing algorithm for train queries). Rather, we start with a wider angle, thinking of the class of problems this relates to. In this case, this is a parsing problem. We can assume this is a context free language, and therefore, can be solved in that scope. With LOP,

$$
\begin{array}{lcl}
query & ::= & routeQuery\ timeQuery \\
 & | & timeQuery\ '\mathbf{,}'\ routeQuery \\
 & | & routeQuery \\
routeQuery & ::= & '\mathbf{from}'\ location\ '\mathbf{to}'\ location \\
 & | & '\mathbf{to}'\ location\ '\mathbf{from}'\ location \\
timeQuery & ::= & '\mathbf{today}' \\
 & | & '\mathbf{tomorrow}' \\
 & | & '\mathbf{on}'\ date
\end{array}
$$

Figure 5: Grammar for transportation queries

we always think of how to *express* the problem or its solution in terms of the problem domain. A common way to express syntax in the problem domain of context-free parsing is the use of the *Backus-Naur Form* (*BNF*) [15]. With this form, we can express the grammar for our query language (Figure 5).

This is obviously a simple language, and a real-life language would be much bigger. However, it is large enough so that implementing an ad-hoc parser for this language in a general-purpose programming language is relatively hard. We would like our parser implementation to be as similar as possible to the grammar. This will make it concise and easy to maintain. To make this happen, we implement the BNF notation as a DSL in Cedalion.

In Figure 5, we can identify five constructs used to define the grammar:

1. The use of a token (quoted strings, $location$, and $date$).
2. The use of a non-terminal symbol.
3. Pattern alternation (|).
4. Concatenation of two patterns (by placing a space between them).
5. A production rule (the ::= operator).

In Cedalion, we can avoid having special constructs for items 2 and 3, making our language smaller. The pattern alternation is given to us for free, by specifying several production rules with the same symbol on their left hand side. This is because statements in Cedalion are related through disjunction by default, each contributing its own solutions. We also do not provide a construct for referring to a symbol, because every symbol in Cedalion is a concept on its own merit. We do, however, add a sixth construct:

6. ε, matching an empty string.

We do so for the completeness of our DSL and for making it viable for other grammars as well, although for the task at hand we could do without it.

4.2 Step 2: Implementing the DSL in Cedalion

Figure 6 provides the definition and implementations of the four required constructs (6, 1, 4, and 5). Statement 6.1 through Statement 6.3 defines the ε pattern, matching an

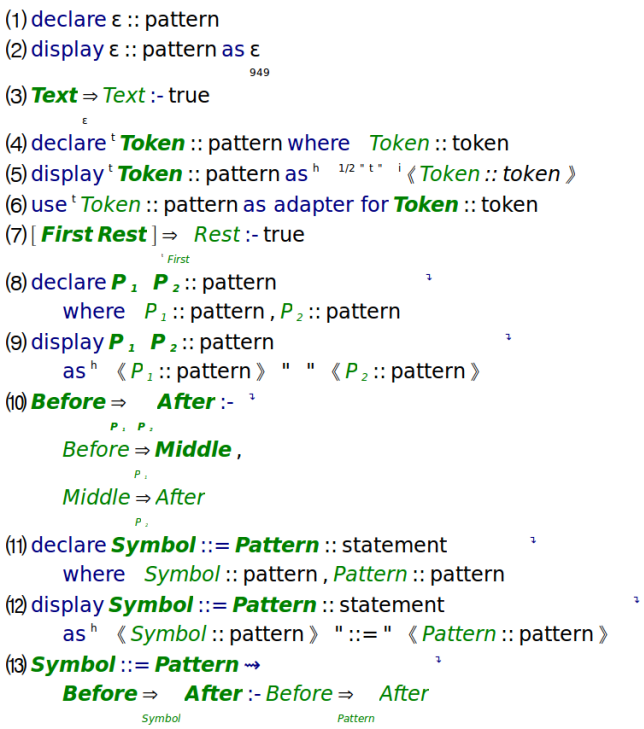


Figure 6: Implementation of the BNF DSL in Cedalion

empty string. Statement 6.1 provides a type signature, defining it as a pattern with no arguments:

$$\textbf{declare}\ \varepsilon :: \text{pattern} \tag{6.1}$$

Statement 6.2 provides the projection definition for ε, defining its representation as the Greek letter Epsilon whose Unicode is 949:

$$\textbf{display}\ \varepsilon :: \text{pattern}\ \textbf{as}\ \underset{949}{\varepsilon} \tag{6.2}$$

This projection definition on the right hand side of the "**as**" in Statement 6.2 immediately affects the way Epsilon is projected on the left hand side in Statement 6.2, as well as in Statement 6.1 and everywhere else.

The blue words in these statements, as well as in the rest of the code, highlight the top level of the statement. This gives the reader a sense of what the statement is, and what the arguments are.

Statement 6.3 defines the semantics of ε (thus implementing it). It defines it as a condition for a state transition. We can look at the problem of top-down parsing as a state transition problem, where the state represents our location on the input, or alternatively, the list of tokens yet to be read. Our initial state is therefore the full list of tokens, and our accepting final state is an empty list. We say that text T matches pattern p if there is a transition p from T to [] (an empty list). We denote that as:

$$T \underset{p}{\Rightarrow} []$$

The ε pattern provides transitions from each state to itself. Statement 6.3 indicates this, by providing a Prolog-like clause:

$$\mathit{Text} \underset{\varepsilon}{\Rightarrow} \mathit{Text}\ \text{:-}\ \text{true} \tag{6.3}$$

The "true" on the right hand side of the :- operator indicates that $\mathit{Text} \underset{\varepsilon}{\Rightarrow} \mathit{Text}$ is true without condition.

Statement 6.4 through Statement 6.7 define the reference to a token. Once again, it starts with a type signature:

$$\textbf{declare}\ {}^{t}\mathit{Token} :: \text{pattern}\ \textbf{where}\ \mathit{Token} :: \text{token} \tag{6.4}$$

This time, this concept has one argument of type "token." Statement 6.5 provides a projection definition, presenting the token with a tiny superscript "t" at its left:

$$\textbf{display}\ {}^{t}\mathit{Token} :: \text{pattern}\ \textbf{as}\ {}^{h\ \frac{1}{2}\ \text{“}t\text{”}\ i}\langle\langle \mathit{Token} :: \mathit{token} \rangle\rangle \tag{6.5}$$

The right hand side of the "**as**" contains a compound visual, consisting of a *horizontal list* (the list with the tiny "h" at its left), containing two elements: a *half size label* (a label visual inside a half size modifier) presenting the tiny "t," and to its right, a place holder for the token. The place holder is indicated using double angle brackets, containing a typed term. The tiny "i" indicates that *Token* is depicted in (green) Italics in the projectional editor.

Statement 6.6 defines this construct as an adapter:

$$\textbf{use}\ {}^{t}\mathit{Token} :: \text{pattern}\ \textbf{as adapter for}\ \mathit{Token} :: \mathit{token} \tag{6.6}$$

In this case, our concept acts as a *pattern* while hosting a *token*. Defining it as an adapter allows Cedalion to consider tokens in the context of patters (e.g., when providing auto complete options), automatically inserting this adapter.

Statement 6.7 defines the semantics of this pattern, as a consumption of a matching token (assuming one exists):

$$[\mathit{First}\ \mathit{Rest}] \underset{{}^{t}\mathit{First}}{\Rightarrow} \mathit{Rest}\ \text{:-}\ \text{true} \tag{6.7}$$

Here too we use true as the condition, but the left hand side term only matches origin states starting with this token.

Statement 6.8 through Statement 6.10 define the concatenation of patterns:

$$\textbf{declare}\ P_1\ P_2 :: \text{pattern}\ \textbf{where}\ P_1 :: \text{pattern}, P_2 :: \text{pattern} \tag{6.8}$$

Note that in Statement 6.9 we use a whitespace as the operator between the two patterns (recall item 4). While unconventional, this is possible in Cedalion, since it uses projectional editing:

$$\begin{array}{l}\textbf{display}\ P_1\ P_2 :: \text{pattern} \\ \quad \textbf{as}\ {}^{h}\langle\langle P_1 :: \text{pattern} \rangle\rangle\ \text{“␣”}\ \langle\langle P_2 :: \text{pattern} \rangle\rangle\end{array} \tag{6.9}$$

Statement 6.10 uses recursive conditions to define the transition based on a concatenation, building a transition from

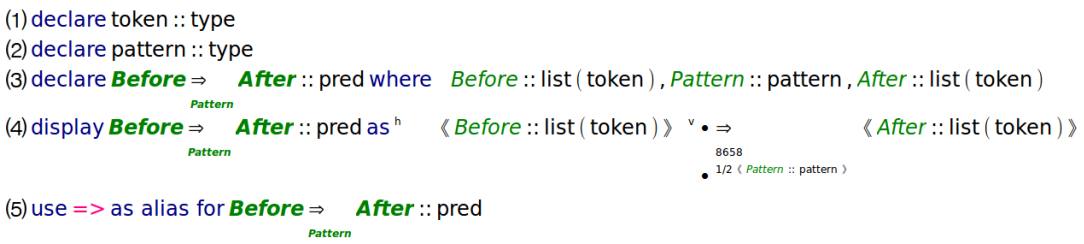


Figure 7: Base definitions for the BNF DSL

Before to *After* if a path exists both from *Before* to *Middle* and from *Middle* to *After*:

$$Before \underset{P_1\,P_2}{\Rightarrow} After \text{ :- } Before \underset{P_1}{\Rightarrow} Middle,\ Middle \underset{P_2}{\Rightarrow} After \tag{6.10}$$

Finally, Statement 6.11 through Statement 6.13 define the production rule. A production rule is a Cedalion statement, and as such, its semantics is provided using the rewrite rule in Statement 6.13. This rewrite rule creates a transition over the pattern at the left hand side of the ::= operator (*Symbol*), if a transition exists for the pattern on the right hand side of the operator (*Pattern*):

$$\begin{aligned} &Symbol \text{ ::= } Pattern \rightsquigarrow \\ &\quad Before \underset{Symbol}{\Rightarrow} After \text{ :- } Before \underset{Pattern}{\Rightarrow} After \end{aligned} \tag{6.13}$$

The definitions in Figure 6 rely on pre-existing definitions of the types *pattern* and *token*, and on the definition of the transition predicate. Figure 7 provides these definitions. The new things here include a vertical layout (indicated by the tiny "*v*" and the vertical list) in Statement 7.4, and an alias definition in Statement 7.5.

4.3 Step 3: Implementing the Parser

Now that we have defined (and implemented) our DSL, we can turn to using it to implement our parser. Figure 8 shows the implementation of the train schedule query parser in our BNF DSL. It starts with declarations of the three token types our parser will encounter: a word (Statements 8.1 through 8.4, indicated by quotes), a date (Statement 8.5) and a location (Statements 8.6 and 8.7). Statements 8.3 and 8.4 defines "word" as an adapter to a string in both the contexts of a token and a pattern, so that strings can be entered in these contexts. Cedalion will then reconcile them by adding this concept as an adapter.

The grammar is given in Statements 8.8 through 8.18. It is similar to the one shown in Figure 5, with some differences. One difference we already mentioned is the fact that we use separate production rules instead of using the | operator. We will need it this way later, when generating output. Another difference is the fact that we provide type signatures for the symbols (Statements 8.8, 8.12, and 8.15). They allow Cedalion to validate the grammar, only based on the type signatures we already provided in Figure 6, and to provide symbol names as options for auto completion. The last

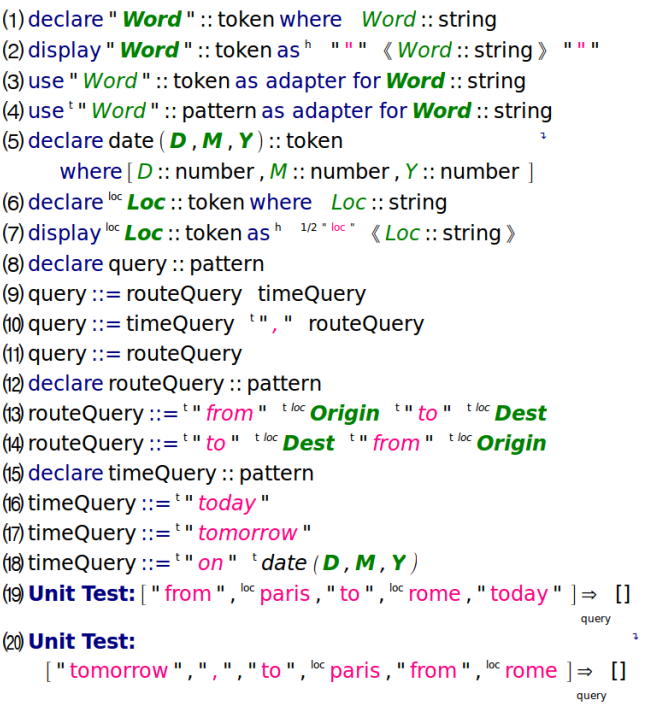


Figure 8: Implementation of the train schedule query parser in the BNF DSL

difference is the fact that tokens appear with a tiny "*t*" to their left. We saw similar markings in projection definitions, e.g., for vertical and horizontal layouts (tiny "*v*" and "*h*," respectively). This is more of a convention than a strict rule, but in Cedalion we prefer having things visible. In this case, we have an adapter (the token pattern) allowing a token to be used as a pattern. The tiny "*t*" allows the user to either select the adapter (by clicking the "*t*"), or to select the token it wraps (by clicking it).

Statements 8.19 and 8.20 provide unit tests for our parser. A Cedalion **Unit Test** construct takes a logic goal as argument, and marks it with an error marker if this goal fails. In this case, we test that both the string "from Paris to Rome today" (Statement 8.19), and "tomorrow, from Rome to Paris" (Statement 8.20) are legal. Note that we provide the input tokenized as the initial state, and the final state is an empty list. We check for a single transition, on pattern query.

```
(1) declare trainQuery :: type
(2) declare routeQueryType :: type
(3) declare timeQueryType :: type
(4) declare Find train Route Time :: trainQuery where   Route :: routeQueryType , Time :: timeQueryType
(5) display Find train Route Time :: trainQuery as h   " Find train " 《 Route :: routeQueryType 》 《 Time :: timeQueryType 》
(6) declare from Origin to Dest :: routeQueryType where   Origin :: string , Dest :: string
(7) display from Origin to Dest :: routeQueryType as h   " from " 《 Origin :: string 》 " to " 《 Dest :: string 》
(8) declare today :: timeQueryType
(9) declare tomorrow :: timeQueryType
(10) declare on D / M / Y :: timeQueryType where   D :: number , M :: number , Y :: number
(11) display on D / M / Y :: timeQueryType as h   " on " 《 D :: number 》 " / " 《 M :: number 》 " / " 《 Y :: number 》
```

Figure 9: Definition of a DSL for querying the train schedule

```
(1) declare query ( Q ) :: pattern where   Q :: trainQuery
(2) query ( Find train Route Time ) ::= routeQuery ( Route )  timeQuery ( Time )
(3) query ( Find train Route Time ) ::= timeQuery ( Time ) t " , "  routeQuery ( Route )
(4) query ::= routeQuery
(5) declare routeQuery ( Route ) :: pattern where   Route :: routeQueryType
(6) routeQuery ( from Origin to Dest ) ::= t " from "  t loc Origin  t " to "  t loc Dest
(7) routeQuery ( from Origin to Dest ) ::= t " to "  t loc Dest  t " from "  t loc Origin
(8) declare timeQuery ( Time ) :: pattern where   Time :: timeQueryType
(9) timeQuery ( today ) ::= t " today "
(10) timeQuery ( tomorrow ) ::= t " tomorrow "
(11) timeQuery ( on D / M / Y ) ::= t " on "  t date ( D , M , Y )
(12) Unit Test: [ " from " , loc paris , " to " , loc rome , " today " ] ⇒  []
                query ( Find train from paris to rome today )
(13) Unit Test: [ " tomorrow " , " , " , " to " , loc paris , " from " , loc rome ] ⇒  []
                query ( Find train from rome to paris tomorrow )
```

Figure 10: Parser implementation with output

4.4 DSL Interoperability

We implemented a parser that allows us to know whether a given list of tokens conforms to some context-free grammar. However, for our imaginary Web application we would like to do more. We would like to *understand* what the list of tokens *means*. For this, we need some way to specify the meaning of train schedule queries.

Figure 9 shows type signatures and projection definitions for the constructs of a DSL for querying the train schedule database. We do not provide the implementation of this DSL here, as it relies on the way the actual train schedule is stored, and is therefore beyond the scope of this example. We purposely defined the projection of its constructs to resemble English text, and for this reason there is a noticeable similarity between some of the projection definitions in Figure 9 and the production rules in Figure 8. With that said, notice that these definitions are independent of one another. They merely describe the same problem.

We now wish to allow our parser to produce query terms in the query DSL defined in Figure 9. Figure 10 shows such a parser, as a refinement to the one we implemented in Figure 8. We re-define our grammar symbols to take arguments. These arguments are the query terms represented by the text that was parsed by these symbols. On the right hand side, these variables appear as logic variables, and they are constructed into query terms on the left hand side. This way, parsing some text will emit an equivalent query term. The unit tests in Statements 10.12 and 10.13 show how the query terms constructed by the parser are similar to the original text. This is a good example of DSL interoperability, where the BNF DSL and the query DSL are agnostic of each other, but still can be used together, with no need for "glue code" in some general-purpose language.

5. Validation: DSL for Bioinformatics

In this sections, we describe a larger case study to demonstrate Cedalion's usefulness. In this case study, we defined in Cedalion a DSL that helped our colleagues at Technion IIT's Department of Biology specify a custom made DNA microarrays for *Protein Binding* [2]. The outcome of their DSL program was sent to production and used in a real experiment in Biology research [1].

5.1 Brief Overview of the Problem Domain

In Biology research, the interaction between proteins and DNA is studied extensively. Such interactions between DNA and a special class of proteins called *transcription factors (TF)* regulates the transcription of genes encoded by the DNA into RNA and eventually into other proteins [3]. Studying these interactions is important to the understanding of physiology and disease processes. However, researching them is far from trivial, due to the complexity of the biological systems involved.

Traditionally, performing *in vitro* (in a test tube) experimentation to test a theory about binding of a protein to a DNA sequence was very expensive and time consuming, since a different experiment was required for every DNA sequence that was tested. Recent development in the field introduced the *Protein Binding Microarray (PBM)* [2]. This is a special kind of *DNA microarray*, a chip containing microscopic wells, each containing a different DNA sequence, with multiple instances in each well (a PBM is a microarray dedicated to binding proteins to DNA sequences). Performing experiments with a microarray allows one experiment to be applied on multiple DNA sequences. Contemporary arrays contain the order of 10^5 different sequences. Experimentation is done by introducing a solution containing the protein under test and possible supplements, such as fluorescent markers to the chip. The protein will bind to a specific DNA sequences, other nonspecific interactions will be eliminated during the wash steps, such that the signal represents only the binding of the TF to a specific DNA sequence. The chip is then scanned using a visible-light scanner, and the fluorescent markers attached to either the protein or the DNA are used to determine which sequences have been bound and which have not. This approach speeds up experimentation tremendously against traditional testing in test tubes, where only one sequence could be tested at a time.

5.2 Problem Statement

The main enabler of this technique is the ability of manufacturers to produce the microarray, along with an order of 10^5 different sequences. This is thanks to a highly automated production process. This possibility leaves the biologist with the need to prescribe these sequences.

Our colleagues at Technion IIT have previously taken an ad-hoc approach to this problem. They created a Java program of about 500 lines-of-code, that provides a text file, containing the desired sequences. This program was made specifically to reflect a certain design, based on a certain biological hypothesis, targeted at a certain experiment. As the research advances, hypotheses change and new microarray designs require changes to that program.

This is where we proposed LOP and Cedalion. With Cedalion we can define a DSL for defining microarray designs, which will allow the generation of the sequence set. The advantage of using a DSL is that the DSL code resembles the way the biologist thinks of the microarray design. This is in contrast to the Java program, where the design is translated to a sequence of commands that produce the list of sequences. We are not naive as to think that a biologist with no programming background would be comfortable implementing such a design using our DSL. However, they would be absolutely comfortable reviewing the DSL code to see that it matches their needs, and even make changes to it.

5.3 Solution Overview

To address the problem of defining DNA microarrays, we first consider a wider problem: defining DNA sequence sets. A DNA microarray consists of different sections (experiment, negative control, positive control, etc.), each is a set of DNA sequences. We provide a DSL for specifying such sets. It is used to specify each section in the DNA microarray.

For each section in the microarray, the user needs to specify three things:

1. the name of this section (for tracking);
2. the DNA sequence set to be used for this section; and
3. the number of sequences to be randomly selected from this set. This will typically be the full size of the set in experiment sections, and an arbitrary small number for control sections.

Generating the microarray is done in two phases. First, use Cedalion to construct files with the full sequence sets for each section. The file name is derived from the microarray name, the section name and the quantity (the number of sequences to be selected). Then a Perl script (<30 lines of code) performs the random selection and provides the final sequences to be used in the microarray. The script also formats the files in a way acceptable by the microarray manufacturer.

Our choice of using a script here aligns with the overall Cedalion philosophy. With this philosophy, a software solution is divided into two parts: (1) the program logic, implemented in Cedalion, preferably by making extensive use of declarative DSLs; and (2) the program's integration to its environment, usually implemented in imperative general-purpose languages. Much of Cedalion's power comes from the fact that it is declarative, with no side effects. For this end, performing some action requires the involvement of some other programming language being able to query Cedalion programs. The idea is to keep the imperative part as small as possible, as part of the DSL implementation and agnostic of the DSL code.

In our case, we need to perform an action (the random decimation of DNA sequences) which does not map well into declarative terms. For this we provide a script. However, this script is agnostic of the DSL code, that is, it has no knowledge that we are dealing with DNA sequences.

The files created by the script can be uploaded to the DNA microarray manufacturer's website when placing an order.

5.4 The Challenge

The challenge in this case study is to provide a DSL that will be intuitive and easy to use by non-programming biologists. We do not expect biologists to be able to build a design from scratch, but we do expect them to be able to review and maybe even modify existing designs. For that, the language must be absolutely intuitive for them. It needs to be declarative, concise, and most importantly, use the common terms of the Bioinformatics world. To accomplish that, we need our LOP environment to provide full freedom in defining this DSL, and not pose any syntactic or other restrictions that would make this language unintuitive to biologists.

In addition, we had a very short window of opportunity to complete the implementation. We had just one week from the time our colleagues knew they needed to define a new microarray design, until they had to submit the list of sequences to the manufacturer. This is due to the long production time of the microarray itself (around one month). In that time frame we needed to implement the DSL, the microarray generation mechanism, the decimation script and help them define their microarray design. Failing to do so would have resulted in our colleagues modifying their Java program to reflect the new design, and using it to generate the sequences.

5.5 A DSL for DNA Sequence Sets

We start by describing a DSL for defining sets of DNA sequences, and its implementation in Cedalion. While we define this DSL for the purpose of DNA microarray definition, DNA sequence sets can be used in a variety of Bioinformatics tasks, such as specifying search patterns for chromosomal searches.

Sets in Cedalion Since our goal is to define sets, expressions in our DSL all represent sets of DNA sequences. Cedalion's bootstrap package provides a "mini DSL" for handling sets. it defines $set(T)$ as the type of sets where all elements are of type T. As in common mathematical notation, testing whether an element X is in set S, and enumerating over the elements of a countable set is done using the $\in$ operator. In Cedalion, the $\in$ operator is a predicate that can be used in the context of logic programming. Cedalion code can define new set constructs by contributing clauses to this predicate. Due to the type system (Section 3.5) the element type must be specified when using the $\in$ operator. Testing whether element X of type T is in set S is therefore done as follows:

$$X \in^{T} S$$

For convenience, Cedalion's "mini DSL" for sets also contains a construct for defining new sets:

$$S \stackrel{def}{=}^{T} D$$

```
(1) declare Nucleotide :: type
        display as " Nucleotide "
(2) declare A :: Nucleotide
        display as " A "
(3) declare A :: set ( list ( Nucleotide ) )
        display as " A "
(4) A def= list ( Nucleotide ) { [ A ] }
```

Figure 11: Definition of the A Nucleotide and Set

This will define set S of type $set(T)$ as equal to D. Here too, we need to specify the element type for type safety. We will use either of these constructs to define our DNA sequence sets. In addition, this "mini DSL" defines concepts such as set unification, intersection, set comprehensions, and a singleton set.

Lists in Cedalion To represent sequences of nucleotides, we use Cedalion lists. Lists are fundamental in Cedalion, as they are in any logic programming language.

Nucleotides and Sequences DNA sequences are sequences of *nucleotides*, marked with the letters A, C, G, and T. We define the type *Nucleotide* to refer to them, and define the four nucleotides as concepts of this type. Figure 11 demonstrates this for the A nucleotide. Statement 11.1 provides the definition of the Nucleotide type. For convenience, we use here a single statement for both the type signature and the projection definition. The same kind of statement is used in Statement 11.2, to define "A" as a Nucleotide. Note that in both statements we define atoms (i.e., concepts with no arguments) but provide a projection definition, mapping it to a label. We do so because we wish to display these concepts differently then their internal representation. For example, the nucleotide A is represented internally as a lower-case "a," to conform with the naming convention for concepts in Cedalion. To conform with the common convention in Bioinformatics, we need a capital "A," and we resolve that by assigning a projection. Similar definitions exist for C, G, and T.

The building blocks of our DSL will be sets representing a single sequence, containing a single nucleotide. Since we have four nucleotides, only four such sets exist, one per each nucleotide. For all practical purposes, these sets are exchangeable with the nucleotide they are related to. We therefore wish to denote them A, C, G, and T as well. Statement 11.3 defines such a concept (for A). To the naked eye, it looks just like the nucleotide A, however, its internal name (visible by clicking it and looking at the text bar), is different. Its type is different too—set of list of Nucleotide, in contrast to just Nucleotide. Statement 11.4 provides the meaning, defining it as a singleton set, containing a sequence (list) with one element: the nucleotide A. Similar definitions are present for C, G, and T.

```
(1) declare Set := Def :: statement
      where  Set :: set ( list ( Nucleotide ) ) , Def :: set ( list ( Nucleotide ) )
      display as h  《 Set :: set ( list ( Nucleotide ) ) 》  " := "  《 Def :: set ( list ( Nucleotide ) ) 》
(2) Set := Def ⇝ Set ≝[def] list ( Nucleotide ) Def
```

Figure 12: Definition of the := operator for DNA sequence sets

```
(1) declare X . Y :: set ( list ( Nucleotide ) )
      where  X :: set ( list ( Nucleotide ) ) , Y :: set ( list ( Nucleotide ) )
      display as h  《 X :: set ( list ( Nucleotide ) ) 》  " . "  《 Y :: set ( list ( Nucleotide ) ) 》
(2) ASet . BSet := { Seq | A ∈[list ( Nucleotide )] ASet ,
                         B ∈[list ( Nucleotide )] BSet ,
                         Seq is the concatenation of A and B of type Nucleotide }
```

Figure 13: Definitions of basic DNA sequence set definitions

```
(1) declare N :: set ( list ( Nucleotide ) )
      display as " N "
(2) N := A ∪ T ∪ C ∪ G
(3) declare W :: set ( list ( Nucleotide ) )
      display as " W "
(4) W := A ∪ T
(5) declare S :: set ( list ( Nucleotide ) )
      display as " S "
(6) S := C ∪ G
```

Figure 14: Definition of some IUPAC codes

In Statement 11.4 we used the $S \stackrel{def^T}{=} D$ statement to provide meaning to a set. It was necessary to enter the type because this concept is polymorphic. In our DSL, however, we are going to define a lot of sets, all with the same element type: $list(Nucleotide)$. We therefore wish to hide the type from the user, given that our users (biologists) are often unfamiliar with type systems, and adding this type (or merely seeing it on the screen) may be a nuisance for them. For this reason, we define the := operator, as a set assignment, specifically defined for nucleotide sequences. Since it always takes the same type (i.e., it is *monomorphic*), the type does not need to appear (as it is inferred by its type signature). Figure 12 provides the definition of the := operator.

Single Nucleotide Sets The *International Union of Pure and Applied Chemistry (IUPAC)* defines standard codes to denote not only a single nucleotide, but also sets of possible nucleotides.[3] The most important one is N, representing *any* of the four nucleotides. Figure 14 shows the definition of three such codes, using the := operator defined in Figure 12, and Cedalion's standard set union operation.

Sequences We define the dot (.) operator to denote concatenation of two sets. $X.Y$ denotes a set of sequences, such that each such sequence can be seen as a concatenation of a sequence in X and a sequence in Y. Formally we define:

$$X.Y = \{s \mid x \in X, y \in Y, s \text{ is a concatenation of } x \text{ and } y\}$$

Figure 13 show the definition of this operator in Cedalion. Note the similarity between Statement 13.2 and the above definition. X^n denotes concatenation of n elements of X, defined in Cedalion as a union of two sets: an empty sequence when $n = 0$, and an element of $X.X^{n-1}$ when $n > 0$. For example, N^3 is the set of all DNA sequences of size 3.

With what we have so far we can define any set of sequences of a given size, restricting the nucleotides in each position. For example, the sequence $N^3.A.T.N^3$ matches all sequences of length 8, where positions 4 and 5 hold A and T, respectively.

Binding We sometimes would like to express sets of sequences that have relations between parts of them. For example, we would like to choose two nucleotides and then repeat them twice, providing a sequence of size four. The sequence $N^2.N^2$ will obviously not perform this task, since each instance of N^2 will be chosen independently. To fix this we *bind* a part of the sequence to a name, and later use this name in one or more places in the sequence. We use the syntax:

$$Y = [X]$$

to denote a binding of an element in X to the name Y so Y can be used later in the sequence. In the above example, the formula:

$$\left(Y = \left[N^2\right]\right).Y$$

will emit the correct result. The parentheses here are only for convenience of showing that the $=$ operator has precedence over the . operator. In Cedalion the parentheses are optional, as the projectional editing resolves the ambiguity problem.

[3] `http://www.bioinformatics.org/sms/iupac.html`

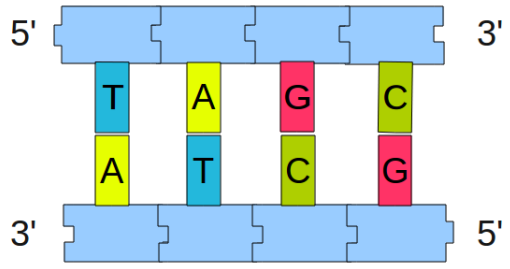


Figure 15: Illustration of a double stranded DNA sequence

Domain-Specific Operators So far we discussed features that relate to sequencing in general. We now add domain-specific operators to the languages, ones that are unique to the field of DNA sequences.

In living creatures, DNA appears double stranded (see Figure 15). Given the sequence of one strand of DNA will allow us to tell the sequence of the other strand, by following two simple steps:

1. Replacing all A's with T's (and vice versa), and all C's with G's (and vice versa). We call the resulting sequence the *conjugate* of the original.
2. Reverse the order. This is because the two strands are positioned in opposite directions.

We use the term *complement* to refer to the sequence of the other strand, denoted X^{comp}, where X is the original strand.

Due to symmetry, A double stranded DNA sequence can be depicted by either strands. If we wish to specify a set of *double stranded* DNA sequences, we need to make sure that, for every sequence in the set, its complement is not in the set. This is unless the sequence *is* its complement, i.e., when $X = X^{comp}$ (a palyndromial sequence).

To facilitate this, we introduce the *restrict* operator. $restrict(X)$ contains for each $x \in X$, either x or x^{comp}, the lower of the two in lexicographic order.

5.6 Generating Microarray Designs

We defined (and implemented) a DSL for defining DNA sequence sets. Now we use them to generate microarray designs. As stated in Section 5.3, we do so in two phases. First, we generate files containing the full factorial of sequences for each section of the microarray design. This is done from within Cedalion. Then, a short Perl script decimates these files, leaving only the ones to be used in the final microarray.

We define a *microarray* statement. This statement has no meaning of its own, that is, it does not have a rewrite rule associated with it. It only provides a place for defining microarray designs. A context menu entry associated with the microarray statement facilitates its generation. The associated procedure generates a file for each section, calculating the file name based on the name of the microarray name, the section name and the quantity (the number of sequences

```
(1) declare spacer1 :: set ( list ( Nucleotide ) )
(2) spacer1 := ( A . C ) ^10
(3) declare spacer2 :: set ( list ( Nucleotide ) )
(4) spacer2 := ( G . T ) ^6
(5) declare feature ( X ) :: set ( list ( Nucleotide ) )
        where   X :: set ( list ( Nucleotide ) )
(6) feature ( X ) := restrict ( N^4 . X . N^4 )
(7) declare fullSequence ( X ) :: set ( list ( Nucleotide ) )
        where   X :: set ( list ( Nucleotide ) )
(8) fullSequence ( X ) :=
        spacer1 . ( Y = [ feature ( X ) ] ) . spacer2 . Y . spacer1
(9) Micro-array our-experiment
        • experiment : fullSequence ( A . C . G . T ) ( 0 )
        • control1 : fullSequence ( A . ( N \ C ) . G . T ) ( 150 )
        • control2 : fullSequence ( A . C . ( N \ G ) . T ) ( 150 )
```

Figure 16: Implementation of our microarray example

desired after decimation). The Perl script uses the name to generate a decimated file based on the specifications in the microarray statement.

5.7 A DNA Microarray Design Example

Figure 16 shows an example of a microarray definition in our DSL. This example is similar in many ways to the one designed by our colleagues, which we do not provide here, since this is still unpublished work on their part.

The microarray design, defined in Statement 16.9, defines three groups: an experiment, and two control groups. They all follow the same pattern, denoted by the *fullSequence*(X) operator, defined in Statements 16.7 and 16.8, where X is a set of sequences that defines the difference between the different groups. The experiment tests $ACGT$, and the control groups replace the C and G with other nucleotides (the $\setminus$ operator depicts set subtraction).

The *fullSequence*(X) operator defines a set of sequences, containing two instances of *feature*(X), separated by spacers ($spacer1$ on both ends, and $spacer2$ in the middle). The two instances are set to be the same, by using binding.

feature(X), defined in Statements 16.5 and 16.6, represents a sequence of twelve nucleotides. The central four are determined by X, wrapped by four nucleotides on each side. Such an experiment is meant to find the combinations of these nucleotides that best bind the protein in question. We *restrict* the set of features, to avoid considering the same double stranded sequence twice.

The spacers, defined in Statements 16.1 through 16.4 are constant sequences, whose purpose is to place the "interesting" features at the right distance from each other and from the edges.

Generating the microarray design is done by selecting "Generate Microarray Design" from the microarray's context menu. Generating the sequences for this example took

a little less than six minutes, about the same time it took to generate the microarray design required by our colleagues. This constructed the full factorial. The decimation (performed by the Perl script) takes a few seconds.

6. Assessment and Discussion

In this section we assess our language-oriented approach to LOP based on our own experience with implementing and using Cedalion and based on the Bioinformatics case study presented in Section 5. We revisit the properties set forth in Section 2 as criteria for this assessment. In our assessment we also consider the following additional DSLs that were implemented in Cedalion. These were not presented here but are available with the Cedalion distribution [4]:

- A general-purpose "DSL" for lazy-evaluation functional programming, featuring the ability to define new expressions (functions), Lambda abstractions and more.
- Two "domain-specific" DSLs for defining processes using CCS notation [21], and for testing their attributes expressed using HML [11, 12].
- Our DSL solution [24] to the challenge assignment defined by the *Language Workbench Competition* of 2011 (LWC'11).
- An example of a software product line (SPL) in Cedalion for calculator software [17].

6.1 Results of the Bioinformatics Case Study

In the Bioinformatics case study we were able to define a DSL in terms the domain experts speak and understand. With some assistance, our Biology colleagues were able to specify with Cedalion the microarray they needed. In fact, before submitting the list of sequences to the manufacturer, they changed the design twice, to accommodate new insights about the problem at hand. The DSL code expressing the design was straightforward enough to allow them to make these necessary modifications by themselves.

The implementation of the DSL took us about one day's work. The implementation of the first microarray design took our colleagues about one hour, with our close assistance. Each modification took them about one or two minutes. The execution of the design, i.e., the generation of the sequences took approximately six minutes. This is in contrast to their Java implementation, where generating sequences typically takes only a few seconds. However, since we only need to generate a design once, one should take the programming time (i.e., the time it takes to create or change a design) into account. According to our colleagues, modifying their Java code to reflect a new microarray design takes them long minutes and even hours in some cases, relative to the minute or two it took to change the DSL code. Taking that into account, the Cedalion implementation performs much better.

The set of sequences produced by the Cedalion-based DSL have been sent to the manufacturer to produce a microarray based on that design. As we are writing these lines, our colleagues are using this microarray for an experiment.

Next we discuss how well Cedalion performed with respect to the properties reviewed in Section 2.

6.2 DSL Definition (Freedom of Expression)

Syntactic Freedom Cedalion uses projectional editing to provide this freedom, breaking away from any restriction posed by a parsing algorithm, or even by the mere use of ASCII text files. Projectional editing allows us to define the syntax for our DSLs in a way that is intuitive for the subject matter experts.

In the Bioinformatics case study in Section 5 we used a notation that was intuitive for biologists. We were able to, for example, use superscript (e.g., X^n) to indicate repetition. If we were to use a parsed language (based on text files), we would not be able to do so, and would need to use an operator that is less intuitive to the domain experts. Since we based our DSL on Cedalion's "mini DSL" for sets, we could use set operators such as $\cup$ in these expressions (recall it was used to define N and other IUPAC codes as depicted in Figure 14).

In the train schedule example in Section 4 we used a double arrow with text below it, to indicate transition. This too could not have been achieved with a parsed language.

Semantic Freedom The diversity of our case studies show that Cedalion DSLs can have diverse semantics. However, the bigger challenge defined by this requirement is to be able to control semantic analysis, and have control over what is considered well-formed DSL code. The case study that put this feature to the test was the LWC [24]. It required a DSL for instances, where each instance conforms with an entity defined by the entity DSL. We used Cedalion's type system to enforce this relationship, e.g., making sure there is a conformance between instances and the entities they are based on. The type system was not powerful enough to enforce cardinality, so we added custom checkers for that task.

6.3 Cost Effectiveness (Economic Freedom)

The most adequate case study to demonstrate DSL cost-effective implementation and usage is the Bioinformatics case study. Here we solved a real-life problem, one that could have been solved in some other way. Estimating the efforts of both ways can allow us to evaluate the cost-effectiveness of using Cedalion for this purpose.

With everything in place, specifying a new design took about one hour. Small modifications to that design took a minute or two each. With the alternative approach, implementing the Java program to generate the sequences took about one day's work, which is comparable with the amount of time we spent setting up the first microarray design in Cedalion. Cedalion's advantage in this starts with the first

modification. It is difficult to assess how long such a modification would have taken for the Java implementation, but it is safe to assume that it would have taken much longer than the minute or two it took with Cedalion, taking into account that the logic for defining the design is spread among approximately 500 lines of code. Changing this code would probably require some level of debugging to achieve the desired results.

6.4 DSL Interoperability (DSLs' Freedom of Association)

Cedalion supports DSL interoperability both semantically and syntactically. Syntactically, it comes from the use of projectional editing. Semantically, this ability is inherent from the fact all DSL code is code in the same programming language, Cedalion.

In the example in Section 4 we demonstrated seamless integration of two mutually independent DSLs: a DSL for parsing based on BNF, and a "business logic" DSL for querying train schedule. They were joined on the DSL code, with no need to modify any of their definitions to allow that. In this integration they were joined both syntactically and semantically.

The DNA sequence set DSL described in Section 5 was based on another DSL, the set "mini DSL." The definitions of most of its concepts were actually made within that DSL, and only few of them were implemented using logic programming clauses. This makes the set operators natural in our DSL, and indeed they were used seamlessly, as in Figure 14 (the $\cup$ operator) and Figure 16 (the $\setminus$ operator).

6.5 Limitations and Threats to Validity

Cedalion presents a viable approach to LOP, and we have shown that our prototyped implementation can be used in a real-life setting to provide a desired outcome. However, we have also witnessed limitations of the approach.

Performance Cedalion is a logic programming language. The programmer enjoys features such as backtracking and term unification without effort. However, these very features of logic programming affect performance. Indeed, in our case study in Section 5, running the Cedalion code took a few minutes, while the Java implementation performing a similar task needed only a few seconds.

However, there are techniques to overcome this limitation. Cedalion can use a faster Prolog engine, which is expected to provide a nice speedup at very little cost. We can also consider combining code generation techniques with logic programming, as a speedup option for mature DSLs. Investigation and implementation of speedup options are left for future work.

Fault Tolerance Cedalion allows for user code to be executed from within the Cedalion workbench, while the code is being edited. This is a powerful tool, but may cause problems if that code contains bugs. A limitation of Cedalion is its inability to withstand all user errors. For example, it is not uncommon in Cedalion to reach non-termination. In such cases, the editor crashes or gets stuck when editing a certain piece of code.

Such problems cannot be avoided altogether. Instead, Cedalion needs to be able to contain these problems. In some cases (e.g., when an exception is thrown), this is easy. However, in some other cases (e.g., non-termination), identifying the problem is hard, and treating it effectively is not easy. We hope that as Cedalion matures we will find ways to address these issues.

Threats to Validity A noticeable threat to validity is the choice of Bioinformatics as a case study. One could claim that this case study coincidentally maps well into Cedalion. However, the variety of examples and other case studies show Cedalion provides consistent result for numerous problem domains, which makes us confident that the results of the case study in Section 5 are indeed representative.

7. Related Work

Cedalion is a host language for internal DSLs that is designed to support flexible DSL syntax and semantics. In comparison, traditional internal DSLs are relatively cost-effective, highly interoperable, but are limited in their syntax and semantics. Lisp and its dialects, for example, have very flexible semantics. However, the syntax of Lisp-based DSLs is usually limited to S-expressions. Implementing readers can help customize the syntax for a single DSL, but does not provide syntactic interoperability.

Most other host languages for internal DSLs are dynamic languages and therefore do not provide static validation for DSL code. However, there are exceptions to the rule, such as typed Scheme [32] and typed Prolog [26], which both implement custom validation through their respective language's macro expansion mechanisms.

Language workbenches share the same goal as Cedalion. The most notable language workbench include the Meta Programming System (MPS) [5], the Intentional Domain Workbench (IDW) [28], and Spoofax [14]. Targeting external DSLs, they mitigate some of their drawbacks by doing one thing similar to internal DSLs: provide a common representation for all DSLs. This common representation often refers to the abstract syntax tree (AST) of the DSL. Although different DSLs use different language constructs, within one language workbench they all have a common notion of what a "language construct" is. These are called *concepts* in MPS, as in Cedalion; and *intentions* in IDW.

This common representation allows the integration of an AST segment for code in one DSL, into the AST of code in another DSL. For this to happen, the DSLs need to agree on some interface (beyond the common representation). The common representation allows semantic interoperability of DSLs. However, DSL interoperability also requires interoperability on the syntactic level (syntactic interoperability).

Language workbenches support that with either scannerless-generalized parsing or projectional editing.

Scannerless generalized LR (SGLR) parsing [33] is an approach that allows parsing of text consisting of a combination of languages (DSLs), when each is defined by a set of production rules (context free grammar), and a set of disambiguation rules (e.g., precedence and associativity). The parser combines all rules (production and disambiguation) from all languages into one grammar. There are no guarantees for it being unambiguous, not to mention belonging to a specific grammar class, such as LALR(1). However, the generalized LR parsing algorithm can use it to parse the file in polynomial time. Ambiguities in parsing the user code are reported as errors. They are then taken care of by either modifying the DSL code (e.g., by placing parentheses or alike to state the structure more explicitly), or by adding disambiguation rules to one of the DSLs. This approach is implemented in the Spoofax [14] language workbench.

Projectional editing [8] is an approach that addresses syntactic interoperability from a different angle. Instead of improving the parsing algorithms to meet the challenges of syntactic interoperability, projectional editing takes the approach of avoiding parsing altogether. In contrast to the traditional approach to programming languages, where the code is being edited in a text editor, saved to a text file, and then parsed to provide a volatile AST, with projectional editing the AST is persistent (mostly referred to as the *model*), and is being edited using a dedicated editor, editing it through a *projection* to a *view*. The projection here is analogous to the *controller* in the model-view-controller (MVC) architecture. The main difference between projectional editing and traditional MVC is in the fact that here the controller (the projection) is defined per-DSL. The most notable projectional language workbenches include MPS [5] and the IDW [28].

Cedalion uses projectional editing for internal DSLs. To our knowledge, this combination has not been tried before. Combining internal DSLs with SGLR parsing has not been tried either, and is a topic for future work.

8. Conclusion

LOP is a paradigm that has been recently attracting more attention. So far it has not been widely adopted in practice, due to the limitations of traditional implementation approaches, namely internal and external DSLs. In a more recent approach, language workbenches offer a promising direction in fighting these limitations, by giving external DSLs a common representation, a feature borrowed from internal DSLs.

In this paper we introduce a novel approach to LOP and an alternative to language workbenches. In contrast to language workbenches, we take internal DSLs and provide them with language workbench features, to make them closer to external DSLs. We introduced Cedalion, an implementation of this approach, as a proof of concept. As evidence that this approach provides a viable alternative to language workbenches, with some trade-offs, we present small examples and a larger case study showing usage of Cedalion to solve a real life problem.

In that case study, a Cedalion-based DSL was successfully used to by biologists in designing a DNA microarray for molecular Biology research. The most glaring advantages of doing so were the readability and the ease of change of the design. The most notable disadvantage was the processing time. This is not a big problem as long as we are at the scale of minutes. When considering the total time it took to generate the sequences, including the code modification time, the Cedalion-based approach showed a significant advantage over the traditional approach. This case study demonstrates how cost-effective LOP can be with our approach.

The Cedalion Eclipse plug-in is implemented as an open source project and publicly available [4].

Acknowledgment

The Bioinformatics case study was a done in collaboration with Itai Beno and Tali E. Haran, Department of Biology, Technion–Israel Institute of Technology, Technion City, Haifa 32000, Israel.

References

[1] I. Beno, K. Rosenthal, M. Levitine, L. Shaulov, and T. E. Haran. Sequence-dependent cooperative binding of p53 to DNA targets and its relationship to the structural properties of the DNA targets. *Nucleic Acids Research*, 39(5):1919–1932, Mar. 2011.

[2] M. F. Berger and M. L. Bulyk. Universal protein-binding microarrays for the comprehensive characterization of the DNA-binding specificities of transcription factors. *Nature protocols*, 4(3):393–411, 2009.

[3] M. L. Bulyk. Protein binding microarrays for the characterization of DNA–protein interactions. *Advances in Biochemical Engineering/Biotechnology*, 104:65–85, 2007. Analytics of Protein–DNA Interactions.

[4] Cedalion. The Cedalion project homepage. Software Engineering Research Lab, The Open University of Israel, 2010. `http://cedalion.sourceforge.net`.

[5] S. Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains onBoard*, 1(2), Nov. 2004.

[6] M. Fowler. Bliki: Fluent interface, Dec. 2005. `http://martinfowler.com/bliki/FluentInterface.html`.

[7] M. Fowler. Language workbenches: The killer-app for domain specific languages, June 2005. `http://www.martinfowler.com/articles/languageWorkbench.html`.

[8] M. Fowler. Bliki: Projectional editing, Jan. 2008. `http://martinfowler.com/bliki/ProjectionalEditing.html`.

[9] S. Freeman and N. Pryce. Evolving an embedded domain-specific language in Java. In *Proceedings of the 21st An-*

nual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06), pages 855–865, Portland, Oregon, USA, Oct. 22–26 2006. ACM Press.

[10] A. Hen-Tov, D. H. Lorenz, A. Pinhasi, and L. Schachter. ModelTalk: When everything is a domain-specific language. *IEEE Software*, 26(4):39–46, 2009. Special issue on Domain-Specific Modeling.

[11] M. Hennessy and R. Milner. On observing nondeterminism and concurrency. In *Automata, Languages and Programming*, number 85 in Lecture Notes in Computer Science, pages 299–309. Springer-Verlag, Berlin, Heidelberg, 1980.

[12] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, Jan. 1985.

[13] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es), 1996.

[14] L. C. Kats and E. Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *Proceedings of the ACM International Conference on Systems, Programming Languages, and Applications: Software for Humanity (SPLASH'10)*, pages 444–463, Reno/Tahoe, Nevada, USA, Oct. 2010. ACM.

[15] D. Knuth. Backus normal form vs. Backus Naur form. *Communications of the ACM*, 7(12):735–736, 1964.

[16] D. H. Lorenz and B. Rosenan. Cedalion: A language oriented programming language. In *IBM Programming Languages and Development Environments Seminar (PLDE'10)*, Haifa, Israel, Apr. 2010. IBM Research.

[17] D. H. Lorenz and B. Rosenan. Code reuse with language oriented programming. In *Proceedings of the 12th International Conference on Software Reuse (ICSR12)*, number 6727 in Lecture Notes in Computer Science, pages 165–180, Pohang, Korea, June 13-17 2011. Springer Verlag.

[18] D. H. Lorenz and J. Vlissides. Designing components versus objects: A transformational approach. In *Proceedings of the 23th International Conference on Software Engineering (ICSE'01)*, pages 253–262, Toronto, Canada, May 12-19 2001. IEEE Computer Society.

[19] S. Mellor, A. Clark, and T. Futagami. Model-driven development. *IEEE software*, 20(5):14–18, 2003.

[20] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37, Dec. 2005.

[21] R. Milner. *A Calculus of Communicating Systems*. Number 92 in Lecture Notes in Computer Science. Springer-Verlag, New York, NY, USA, 1980.

[22] A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial intelligence*, 23(3):295–307, 1984.

[23] B. Rosenan. Designing language-oriented programming languages. In *Companion to the ACM International Conference on Systems, Programming Languages, and Applications: Software for Humanity (SPLASH'10)*, pages 207–208, Reno/Tahoe, Nevada, USA, Oct. 2010. ACM. Student Research Competition, 2nd prize.

[24] B. Rosenan. Cedalion submission to the language workbench competition of 2011. In M. Völter, E. Visser, S. Kelly, A. Hulshout, J. Warmer, P. J. Molina, B. Merkle, and K. Thoms, editors, *Language Workbench Competition*. 2011. `http://www.languageworkbenches.net/lwc11-cedalion.pdf`.

[25] C. Sassenrath. The REBOL scripting language. *Dr. Dobb's Journal: Software Tools for the Professional Programmer*, 25 (7):64–68, 2000. `http://rebol.com`.

[26] T. Schrijvers, V. S. Costa, J. Wielemaker, and B. Demoen. Towards typed Prolog. In *Proceedings of the 24th International Conference on Logic Programming (ICLP'08)*, pages 693–697, Udine, Italy, 2008. Springer-Verlag.

[27] C. Simonyi. The death of computer languages, the birth of intentional programming. Technical Report MSR-TR-95-52, Microsoft Corporation, 1995.

[28] C. Simonyi, M. Christerson, and S. Clifford. Intentional software. *ACM SIGPLAN Notices*, 41(10):451–464, 2006.

[29] Z. Somogyi, F. Henderson, and T. Conway. Mercury, an efficient purely declarative logic programming language. *Australian Computer Science Communications*, 17:499–512, 1995.

[30] T. Stahl and M. Völter. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[31] C. Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 2nd edition, 2002. With Dominik Gruntz and Stephan Murer.

[32] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 395–406, San Francisco, California, USA, Jan. 2008. ACM.

[33] E. Visser. Scannerless generalized-LR parsing. Technical Report P9707, University of Amsterdam, Programming Research Group, Department of Computer Science, Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands, Aug. 1997.

[34] M. P. Ward. Language-oriented programming. *Software-Concepts and Tools*, 15(4):147–161, 1994.

[35] J. Wielemaker. An overview of the SWI-Prolog programming environment. In F. Mesnard and A. Serebrenik, editors, *Proceedings of the 13th International Workshop on Logic Programming Environments (WLPE'03)*, pages 1–16, Mumbai, India, Dec. 2003. Report CW371, Katholieke Universiteit Leuven, Nov. 2003.

[36] XLR. XLR: Extensible language and runtime, 2008. `http://xlr.sourceforge.net/concept/XL.html`.