CRStL: A Declarative Language for the Encoding of Proof Techniques

Dominik Dietrich[1] Ewaryst Schulz[2]

FR 6.2 Informatik
Saarland University
Saarbrücken
Germany

Abstract

We propose the language **CRStL** to formulate mathematical reasoning techniques as proof strategies in the context of the proof assistant ΩMEGA. The language is arranged in two levels, a query language to access mathematical knowledge maintained in development graphs, and a strategy language to annotate the results of these queries with further control information. The two-leveled structure of the language allows the specification of proof techniques in a very declarative way. We give examples to illustrate the use and semantics of **CRStL**.

Keywords: tactic language, query language, proof search, proof planning

1 Introduction

Unlike widely used computer-algebra systems, mathematical assistance systems have not yet achieved considerable recognition and relevance in mathematical practice. One significant shortcoming of the current systems is that they are not fully integrated into or accessible from standard tools that are already routinely employed in practice, like, for instance, standard mathematical text-editors. Integrating formal modeling and reasoning with tools that are routinely employed in specific areas is the key step in promoting the use of formal logic based techniques.

Therefore, in order to foster the use of proof assistance systems, we integrated the theorem prover ΩMEGA [11] into the scientific text-editor TEXMACS [13]. The goal is to assist the author inside the editor while preparing a TEXMACS document in a publishable format. The vision underlying this research is to enable a document-centric approach to formalizing and verifying mathematics and software, that is, we are aiming at human readable,

[1] Email: dietrich@ags.uni-sb.de
[2] Email: schulz@ags.uni-sb.de

human writable, and machine checkable documents. In our current implementation theories can be formalized within the text editor and proofs can interactively be constructed by the application of basic proof operators, proof strategies, which are similar to tactics, or the introduction of proof sketches. A significant shortcoming in the current version of our system is that proof strategies cannot be specified within the document. Rather, they need to be programmed in the underlying programming language of the proof assistant, which is in the case of ΩMEGA the programming language Lisp. It is clear that for complex and efficient proof strategies a complete programming language is necessary. However, small parts of proofs can often be automated by special purpose proof strategies, which could in principle simply be specified by the user. However, requiring the use of the underlying programming language often prevents the user to take this option because he is unfamiliar with the language. Even for experienced users it is often too time consuming to design a special purpose proof strategy in the underlying programming language.

Consider for example the following simple theorem about binary relations:

$$(R \cup S) \circ T = (T^{-1} \circ R^{-1})^{-1} \cup (T^{-1} \circ S^{-1})^{-1} \tag{1}$$

Suppose further that we have already proved a theorem "Inverse composition"

$$\forall R, S.(R \circ S)^{-1} = S^{-1} \circ R^{-1}, \tag{2}$$

a theorem "Distributivity of $\circ$ over $\cup$":

$$\forall R, S, T.(R \cup S) \circ T = (R \circ T) \cup (S \circ T) \tag{3}$$

and a theorem "Doubleinverse identity":

$$\forall R.R = (R^{-1})^{-1} \tag{4}$$

A possible approach to proof (1) would be to expand all the definitions involved in (1) and then to try to prove the resulting formulas using propositional logic rules, which is already implemented in ΩMEGA as a generic strategy. However, there is a more elegant way. It is easy to see that the previous proved theorems (2), (3) and (4) can directly be used to prove the problem within a few steps. Writing a proof strategy for this specific case in the underlying programming language is laborious; a more lightweight interface would be beneficial.

In this paper we present the declarative language **CRStL** [3] for the specification of proof strategies in ΩMEGA. Similar to $\mathcal{L}_{tac}$ [8] the language is intended to bridge the gap between the predefined proof operators and the programming language of the proof assistant. Our language differs from standard languages for tactics with respect to the following aspects: (1) We consider the theory environment in which the proof construction takes place as a data base and provide an explicit query mechanism to retrieve knowledge items from this environment, in particular proof operators. The retrieved knowledge items can be annotated with additional control information such as matching conditions. Building proof strategies

[3] **C**ontrol**R**ule**St**rategy **L**anguge, pronounce "crystal"

on top of this query mechanism allows the specification of the proof operators to be used within a proof strategy by their properties. This way proof strategies automatically adapt to new contexts when the theory changes. (2) We provide language constructs to select subgoals for subsequent strategy executions by their properties. Thus a proof strategy does not depend on the order in which the subgoals were introduced. (3) In addition to procedural descriptions of proof operators we support also the specification of goal descriptions which will then be used as solution conditions in the corresponding strategy.

The paper is organized as follows: In Sec. 2 we set the context of this work by describing proof construction and knowledge management in the ΩMEGA system. Sec.3 gives an overview of the strategy language **CRStL** which allows the specification of proof strategies in ΩMEGA. We conclude the paper with a discussion in Sec. 4.

2 ΩMEGA's Reasoning Framework

ΩMEGA provides a generic framework for proof construction which consists of (1) the development graph to store the mathematical knowledge structured in theories such as definitions, and axioms (2) the TASKLAYER to maintain a proof attempt and to perform basic proof steps (3) a proof planner to perform a search based on the notion of proof strategies and control rules. In this section we give an overview of these components.

2.1 Knowledge Management in the Development Graph

The knowledge of the ΩMEGA system is organized in axiomatic theories that are built on top of each other by importing knowledge from lower theories via theory morphisms. This organization is based on the notion of *development graphs* (see [4] for details).

Each theory in MAYA's development graph contains standard information like the signature of its mathematical concepts and corresponding axioms, lemmas and theorems. In addition to these notions, the development graph allows the specification of other kinds of knowledge, which are not necessarily affecting the semantics of a theory but which, for instance, provide valuable information to proof procedures. An example is ordering information for the function symbols in the signature of a theory, which can be exploited by simplification procedures. Knowledge can either manually be added to a knowledge kind, or automatically be classified by classification functions. For example, one either specifies a formula to be a definition, or relies on predefined heuristics for definition detection.

Each knowledge item is attached to a specific theory and is visible in all theories that link to this theory. The links can use morphisms to transform the structures from the source theory, therefore the knowledge items are transformed also along these morphisms . For instance, a morphism that renames a function f into a function g transforms an ordering information that $f > h$ (for some function h of the signature) into the ordering information $g > h$. The default behavior for knowledge transformations is simply to transform all terms which occur inside the knowledge item.

2.2 *The* TASKLAYER

The central component for proof construction in ΩMEGA is the TASKLAYER. It is based on the CORE-calculus [2] that supports proof development at the *assertion level* [10], where proof steps are justified directly by definitions, axioms, theorems or hypotheses (collectively called *assertions*) omitting basic logic inference applications for changing the structure of the formula to match exactly the expected form of the assertion.

Tasks. At the TASKLAYER, the main entity is a task, a multi-conclusion sequent $F_1,\ldots,F_j \vdash G_1,\ldots,G_k$, expressing a subgoal to be proved. Initially there is only a single task consisting of the conjecture to be proved. A proof attempt is represented by an *agenda*, consisting of a set of tasks, a global substitution which instantiates meta-variables, and contextual information. Tasks are reduced to subtasks by applying proof operators, called *inferences*. If a task has been reduced to an empty set of subtasks, it is called *closed*, otherwise it is called *open*. Tasks and subformulas can be assigned named foci of attention that are maintained within the actual proof.

Inferences. The basic operators for proof construction are so-called *inferences*. Intuitively, an *inference* is a proof step with multiple premises and conclusions augmented by (1) a possibly empty set of hypotheses for each premise, (2) a set of *application conditions* that must be fulfilled upon inference application, (3) a set of *completion functions* that compute the values of premises and conclusions from values of other premises and conclusions. Usually, inferences encode the operational behavior of domain specific assertions. However, they can also encode proof planning methods or calls to external special systems such as a computer-algebra system, an automated deduction system or a numerical calculation package (see [9,3] for more details).

Rather than working with a fixed set of predefined inferences, the set of inferences is continuously extended whenever a new theorem has been proved. The technique to obtain such inferences automatically from assertions follows the introduction and elimination rules of a natural deduction calculus (see [3] for details). For the purpose of this paper it is important to note that one formula can result in several inferences. Consider for instance the domain of set theory and the definition of $\subseteq$:

$$\forall U,V.U \subseteq V \Leftrightarrow (\forall x.x \in U \Rightarrow x \in V) \tag{5}$$

That assertion gives rise to two inferences:

$$\frac{\begin{array}{c}[x \in U]\\ \vdots\\ p : x \in V\end{array}}{c : U \subseteq V}\,\textit{Def-}\subseteq \qquad \textbf{Appl. Cond.: } x \textit{ new for } U \textit{ and } V \tag{6}$$

$$\frac{p_1 : U \subseteq V \qquad p_2 : x \in U}{c : x \in V}\,\textit{Def-}\subseteq \qquad \textbf{Appl. Cond.: } - \tag{7}$$

Reading bottom up, the first inference can be paraphrased as follows: In order to show $U \subseteq V$, we can assume $x \in U$ and have to show $x \in V$ for a fresh variable x. U and V are meta-variables that need to be instantiated during the matching of the inference. However, we can also read (and apply) the inference top down: If we know that $x \in V$

under the assumption $x \in U$ for arbitrary x, then we can conclude $U \subseteq V$.

2.3 Proof Automation

ΩMEGA provides a proof planning framework to encode and apply common patterns of reasoning, called *strategies*. Intuitively a strategy performs a heuristically guided search using a dynamic set of inferences (as well as other strategies), and control rules which determine the planner's behavior at choice points until a specified termination condition succeeds or the search space has been completely traversed. A strategy application results in a nonempty list of solutions encoded in form of agendas or fails. Technically a strategy is a function from an agenda to a list of agendas.

In a nutshell, a strategy execution corresponds to the following loop, which is executed until a specified termination condition is satisfied or the search space is completely traversed: (1) Selection of an agenda A (2) Selection of an open task T from A (3) Selection of the proof operators to be applied to T (4) Instantiation of the proof operators with respect to T under some additionally specified constraints, resulting in a list of so-called *partial argument instantiations* (pais) (5) Selection and application of a subset of the pais, resulting in new agendas. Those new agendas satisfying a solution condition are collected and not further modified. Backtracking occurs if no pais can be produced in step (4). However, there is also the possibility to explicitly backtrack, e.g., if a certain depth has been reached.

Up to now, the only possibility to formulate strategies in ΩMEGA is to use a language based on lisp s-expressions, called POST. This language doesn't provide (1) an access to the structured knowledge stored in the development graph other than symbols, axioms or theorems and (2) a pattern matching facility for term matching and filtering.

3 The Language and its Components

This section introduces the main components of the language **CRStL**. Our goal is to provide a language which permits the user to implement new proof techniques, and thus to extend the systems automation capability, without requiring the knowledge of the systems underlying programming language and its automation API. We are not aiming at a complete programming language, rather at a small, extensible, intuitive, but restricted language for automating small parts of proofs. We want to cover the full spectrum from declarative proof strategies, i.e., specifying what to achieve and what knowledge to be used, to full procedural proof strategies, i.e., specifying what proof operators to apply in which order. Our main requirements are:

- Structured access to the mathematical knowledge stored in the development graph to specify a subset of the proof operators to be used within a proof strategy, similar to SQL or XPath.
- Dynamic binding of variables to parts of the queried structures, especially terms, for later use in constraints or ordering statements.
- Abstraction over the conversion of knowledge to different formats, in particular the conversion of axioms and theorems to inferences.

- Easy access to terms and tasks to express their structural properties, e.g., for specifying constraints to restrict the search space, as well as solution conditions for strategies.
- Introduction of explicit backtracking conditions.
- Restriction of the instantiation possibilities for proof operators.
- Specification of orderings at choice points.
- Support of new user defined predicates and functions.

The language is arranged in two levels, a query language to access the mathematical knowledge, and a strategy language which makes extensive use of these queries and annotates the result of a query with further control information to build a proof strategy. Table 1 summarizes the language **CRStL** in a BNF like notation.

To get a first feeling for the language consider the following proof strategy "Special Purpose" for the motivating example, shown in Listing 1. "Special Purpose" essentially consists of four parts: a specification of a backtrack event, a repeat loop, a use expression and a select expression. The easiest way to understand the strategy is to read the defining expression from the inside to the outside. The select expression specifies two theorems to be selected from the theorems of the current theory. The use expression performs the conversion of the theorem names to inferences. The result of the query is a set of two inferences, representing the four rewrite rules

$$(R \cup S) \circ T \to (R \circ T) \cup (S \circ T) \tag{8}$$
$$(R \circ T) \cup (S \circ T) \to (R \cup S) \circ T \tag{9}$$
$$(R \circ S)^{-1} \to S^{-1} \circ R^{-1} \tag{10}$$
$$S^{-1} \circ R^{-1} \to (R \circ S)^{-1} \tag{11}$$

The **solve** construct specifies that the goal of the proof strategy is to solve the task to which it is applied to. This already includes backtracking if none of the rewrite rules is applicable in a situation. Finally, **backtrack-if** enriches the backtracking behavior of the proof strategy such that a backtrack event is invoked whenever the function call `(> stratdepth 3)` evaluates to true. In this expression `stratdepth` is a predefined variable which is bound at runtime to the depth of the search tree by the proof strategy. The **define-strategy** binds the proof strategy to the name "Special Purpose". However, it is also possible to directly invoke a strategy expression.

```
1 define-strategy "Special Purpose"
2   solve
3    use
4     select "Distributivity of ∘ over ∪",
5            "Inverse composition"
6     from current.theorems
7   backtrack-if
8    (> stratdepth 3)
9 end;
```

CRStL Listing 1: Special purpose strategy for the example

```
<select>        ::= select <selector> from <source> <wherecond>?
                |   <select> union <select>

<selector>      ::= *
                |   term
                |   (name,)* name
<source>        ::= theoryname
                |   theoryname.knowledge

<infexpr>       ::= use <select> <infcond>? <wherecond>
                |   <infexpr> union <infexpr>
                |   <infexpr> intersection <infexpr>
                |   <infexpr> difference <infexpr>
<infcond>       ::= as <infdirection>
<wherecond>     ::= where <function_call>
<infdirection>  ::= forward | backward | close

<listexpr>      ::= <infexpr> | <stratexpr>+

<stratexpr>     ::= first <listexpr>
                |   solve <stratexpr>
                |   repeat <stratexpr> <untilcond>?
                |   ( <stratexpr> )
                |   try <stratexpr>
                |   <stratexpr> then <stratexpr>
                |   name
                |   <stratexpr> backtrack-if <cond>
                |   cases <case>+ end;
                |   <stratexp> thenselect cases <case>+ end;

<case>          ::= <cond> -> <stratexpr>

<untilcond>     ::= until <cond>

<cond>          ::= <matcher>
                |   <function_call>

<defstrat>      ::= define-strategy name <parameter>? <stratexpr> end;

<parameter>     ::= parameter (name,)* name;
<matcher>       ::= <matchhead> <matchcond>?
<matchhead>     ::= <sequent> | var
<matchcond>     ::= where  <function_call>
<sequent>       ::= (<termpattern>,)* <termpattern> |- <termpattern>
<namedterm>     ::= <term> | name:<term> | *
<termpattern>   ::= <namedterm> | [<namedterm>] <termqualifier>?
<termqualifier> ::= + | -
```

Table 1
Syntax of **CRStL**

The previous example showed how we can realize a simple depth limited search strategy within our framework. In general the user has to find a trade-off between the restriction of the search space and the simplicity of the search strategy by further control constructs. As the search space is very small in our example, we decided in favor of a very simple proof strategy. Note that in our language function calls, such as the depth limiter from the previous example, build the interface to the underlying programming language and can comprise manipulations of arbitrary lisp objects. Moreover, function calls have access to a number of predefined variables, such as the execution time, the agenda, the theory, or the search depth. We now explain the language constructs in more detail.

3.1 The select *Statement*

The select statement is used to select knowledge items from a specified theory. It consists of three parts, a **selector** part, a **from** part, and a **where** part.

The From Part. The **from** part specifies the knowledge source and the theory of the development graph from which the knowledge is retrieved. The theory and the corresponding knowledge source can be accessed by their names. Moreover, we support a number of predefined keywords, e.g., to access the current theory, or the base theory. An overview of the available keywords is shown in Table 2.

From a global perspective one can divide the knowledge into two parts: *direct knowledge*, and *indirect knowledge*. *Direct* knowledge is knowledge which has the form of a proof operator, i.e., an inference or a proof strategy, or can be converted to such. For that purpose knowledge transformation functions need to be specified. For example the function which transforms a name to an axiom simply returns the first axiom which has the specified name. The transformation function from formulas to inferences determines exactly those inferences which were synthesized from the specified formula. An example of a direct knowledge item is an axiom. In this case the conversion of the formula results in all inferences which were obtained from this formula. Thus in case of the definition of subset (5) these are the inferences shown in (6), and (7).

Indirect knowledge is knowledge which cannot be converted to a proof operator, but which can be used at choice points. An example for indirect knowledge is a symbol ordering. It cannot be converted to a proof operator, however it can indirectly be used to restrict the applicability of the proof operators.

Note that new knowledge kinds can easily be added to the development graph and are directly available in the query language.

The Selector Part. The **selector** part works on the knowledge kind specified in the **from** part and can be used to further narrow down the set of knowledge items returned by the query. In the simplest case the **selector** part consists of a single $*$. In this case all knowledge items are returned. The **selector** part may also consist of a set of names, in which case only those knowledge items are returned for which there is a name in the specified list of names. Finally there is the possibility to specify a term pattern. In this case only those knowledge items which correspond to a formula that matches the pattern are returned. The term pattern shares the variables with the proof context in which the query is evaluated. Free variables are interpreted as meta-variables to be instantiated by the query. These instantiated variables are passed to the **where** part and can be used there for the

all	all theories reachable from the current context
current	only the local knowledge of the current theory
base	only the underlying logic
top(n)	only the theories reachable from the current theory in n steps
“name”	only the local knowledge of the theory given by name
axioms	the axioms
theorems	already proved theorems
formulas	axioms and proved theorems
definitions	axioms which were classified as definitions
inferences	inferences, user defined and derived from axioms
strategies	user defined strategies
knowledge	stands for any knowledge

Table 2
Available source keywords

specification of additional constraints. We found out that it is convenient for the matching to remove all leading quantifier of formulas corresponding to knowledge items.

The Where Part. The **where** part can be used to specify additional constraints the knowledge items of the query have to satisfy. A variable binding which stems from a pattern matching in the **selector** part is available for evaluation of the expression in the **where** part. Listing 2 shows an example of a select expression consisting of a **selector** part, a **from** part, and a **where** part. The query returns those axioms from the current theory which are equations (after removing the leading quantifiers) $lhs = rhs$ and binds the lefthand side of the equation to the variable *lhs*, and the righthand side of the equation to the variable *rhs*. In the **where** part, it is checked whether *lhs* is greater as *rhs* using the LPO with the symbol ordering stored in the development graph as measure for the terms.

```
1 select lhs=rhs from current.axioms
2   where (greaterlpo lhs rhs
3     (select * from current.ordering))
```

CRStL Listing 2: Example of a select expression which binds variables *lhs* and *rhs* for the specification of additional constraints

3.2 The use *Statement*

The use statement can be invoked directly after a select statement and performs two tasks: (1) Knowledge items are transformed to proof operators using the installed transforma-

tion functions. If no conversion is possible, the user is informed that the query cannot be correctly interpreted. (2) The obtained proof operators are augmented by further control information. For inferences there is the possibility to restrict their application direction using the **as** keyword. **Backward** restricts the applicability of inferences to those where all conclusions are instantiated. **Forward** specifies that no conclusion must be instantiated, and **close** that all premises and all conclusions must be instantiated.

Moreover, the automation API can be accessed via the underlying programming language using a **where** condition. As an example consider the definition of a standard simplification proof strategy shown in Listing 3.

```
1 define-strategy "Simplification"
2   repeat
3     first
4       use select lhs=rhs from current
5         where (greaterlpo lhs rhs ordering) as forward
6       union
7       use select lhs=rhs from current
8         where (greaterlpo rhs lhs ordering) as backward
9 end;
```

CRStL Listing 3: Standard simplification encoded as proof strategy

3.3 Strategy Constructors

The final step in the specification of a proof strategy consists of augmenting the specified list of proof operators by control information needed for their execution. Essentially this consists of a specification of a condition for success and termination. Moreover, there is the possibility to specify a condition for failure, i.e., to directly invoke backtracking. An overview of the strategy constructors is shown in Table 3. The operators can be classified into the three categories **selector**, **iterator** and **combinator**.

Selectors. A list of given proof operators augmented by control information can be instantiated, resulting in a list of pais satisfying a specified set of constraints. **Selector** expressions determine how the pais are produced, and which pai to choose among the set of produced pais. For example, the **first** selector starts to instantiate the proof operators and applies the first which is applicable and satisfies the specified constraints. Another selector is the **cases** selector. It consists of a list of condition action pairs, where the condition is encoded in form of a matcher or a function call. It executes the first action whose condition is satisfied. Similar to the matching in select-expressions bound variables are passed to subsequent expressions.

```
1 cases
2   * |- ~x -> use select "Contradiction" from base.inferences
3   default -> use select "same" from base.inferences
4 end;
```

CRStL Listing 4: The **cases** construct

first	selector	applies the first proof operators that succeeds.
solve	iterator	applies the strategy until it fails or a solution was found.
repeat	iterator	applies the strategy until it fails or until the condition evaluates to true.
try	combinator	applies the strategy and doesn't fail if strategy fails.
then	combinator	applies the first strategy and then the second strategy. Fails if the first strategy fails. If second strategy fails then first strategy backtracks internally and the second strategy is invoked again.
backtrack-if	combinator	adds a backtrack event specified in condition to the strategy.
cases	selector	applies the first strategy whose condition evaluates to true or pattern matches. On multiple matches for a single pattern a backtrack point will be defined for each match.
thenselect	combinator	executes a strategy and applies specified strategies to the resulting tasks

Table 3
Strategy Constructors with corresponding classification

Listing 4 shows an example of the **cases** selector. The matcher `* |- ~x` encodes the condition that the goal of the current task is a negated formula. In this case, proof by contradiction shall be performed. As default case the inference "same" is applied.

Iterators. An **iterator** encodes a loop together with a default backtracking behavior. So far, we support two iterators, **solve** and **repeat**. **solve** tries to close the task to which it was applied to by repeatedly applying the specified proof operators. If none of them is applicable, it backtracks. It fails if the complete search space is traversed. **repeat** applies the specified proof operators until none of them is applicable anymore. Additionally, a termination condition can be specified using the **until** keyword.

Combinators. Combinators combine strategy expressions which they take as arguments. So far, we support four strategy combinatory, **try**, **then**, **backtrack-if**, and **thenselect**. Whereas the first two have their standard meaning, the latter need further explanation. **Backtrack-if** adds a failure condition to a specified condition and thus explicitly invokes backtracking. **Thenselect** analyses tasks resulting from a strategy application and maps them to new strategy applications. Note that by using conditions the mapping of subsequent proof strategies to tasks does not depend on the order of the tasks.

An example using the **thenselect** combinator is shown in Listing 5. The proof strategy "Induction" tries to perform an induction on the current goal. Successfully applied induction results in a list of subtasks containing step- and base-cases. It then applies a standard simplification to all base cases, and rippling to all step cases.

```
1  define-strategy "Induction"
2    first
3      use select axiom from current.axioms
4          where (isinductionaxiom axiom)
5      as backward
6    thenselect cases
7      task where (isbasecase task) -> "Simplification"
8      default -> "Rippling"
9    end;
10 end;
```

CRStL Listing 5: Induction proof strategy illustrating the **thenselect** construct

As we have already seen in the examples the matching facility is quite powerful. In addition to match top-level formulas, we also allow the matching of arbitrary subformulas. This is in particular useful because ΩMEGA allows the application of inferences deeply inside formulas. We use the intuitive notation `[term]` that `term` can be a subterm. Moreover, we support the restriction of subformulas to specific polarities (c.f. [14]), where we use + to indicate a subformula with positive polarity (intuitively a goal to be shown) and - to indicate a subformula with negative polarity (intuitively a fact). The proof strategy shown in Listing 6 takes a formula as parameter and tries to derive a task in which the formula occurs with negative polarity and no dependencies.

```
1  define-strategy "Fact"
2    parameter formula;
3    repeat
4      use select * from current.inferences as forward
5    until *,[formula]- |- *
6      where (not (proofobligations (pos formula)))
7    backtrack-if (greater stratdepth 3)
8  end;
```

CRStL Listing 6: Proof strategy deriving a specified formula

4 Discussion and Future Work

In this paper we have presented the language **CRStL** for the encoding of proof techniques in form of proof strategies in the proof assistant ΩMEGA. With **CRStL** we aim to bridge the gap between the predefined proof operators and the underlying programming language of ΩMEGA. **CRStL** supports declarative and procedural descriptions of proof strategies, as well as a mixture of them. The main idea was to see the specification process of a proof strategy as a two-staged process, consisting of (1) the selection of proof operators and (2) augmentation of these proof operators with control knowledge.

The selection process is inspired by the query languages SQL, OQL and XPath, which are standard for querying structured knowledge such as relational/object-oriented data bases or XML. Indeed, we see the development graph as an object-oriented hierarchical

data base, object-oriented in the sense that the different knowledge kinds correspond to different classes and the theories serve as a hierarchical structuring mechanism.

Searching and retrieving knowledge from mathematical data bases has been studied in the context of Helm [1] and Mizar [12]. Helm focuses on the interaction with different mathematical repositories over the web and Mizar uses the MML Query system[6] also for presentational purposes in MMLQT[5], such as for instance XPath is used in XSLT. Most similar to the strategy constructors, in particular the matching constructs, is the language $\mathcal{L}_{tac}$ [8]. The augmentation of proof operators with control knowledge can also be found in the ELAN system [7].

Future work comprises the refinement of our constructs. For example we plan to add sorting possibilities to the language to express preferences among a set of inferences more naturally. Moreover, so far the language supports only proof strategies working on a single agenda. As ΩMEGA already supports the management of multiple proof attempts in parallel, it is a natural step to extend the language to cope with multiple agendas. It would also be interesting to integrate lemma speculation in **CRStL**.

References

[1] Andrea Asperti, Luca Padovani, Claudio Sacerdoti Coen, Ferruccio Guidi, and Irene Schena. Mathematical knowledge management in helm. *Annals of Mathematics and Artificial Intelligence*, 38(1-3):27–46, 2003.

[2] S. Autexier. The CoRe calculus. In R. Nieuwenhuis, editor, *Proceedings of CADE-20*, LNAI 3632, Tallinn, Estonia, july 2005. Springer.

[3] S. Autexier and D. Dietrich. Synthesizing proof planning methods and oants agents from mathematical knowledge. In J. Borwein and B. Farmer, editors, *Proceedings of MKM'06*, volume 4108 of *LNAI*, pages 94–109. Springer, august 2006.

[4] S. Autexier, D. Hutter, Till Mossakowski, and Axel Schairer. The development graph manager MAYA. In Hélène Kirchner and C. e Ringeissen, editors, *Proceedings 9th International Conference on Algebraic Methodology And Software Technology (AMAST'02)*, volume 2422 of *LNCS*. Springer, September 2002.

[5] Grzegorz Bancerek. Information retrieval and rendering with mml query. In *Proceedings of MKM'06*, pages 266–279. Springer-Verlag, 2006.

[6] Grzegorz Bancerek and Piotr Rudnicki. Information retrieval in mml. In *Proceedings of MKM'03*, pages 119–132, London, UK, 2003. Springer-Verlag.

[7] Peter Borovansky, Claude Kirchner, Hne Kirchner, and Christophe Ringeissen. Rewriting with strategies in ELAN: A functional semantics. *International Journal of Foundations of Computer Science*, 12(1):69–95, 2001.

[8] David Delahaye. A Proof Dedicated Meta-Language. In *Proceedings of Logical Frameworks and Meta-Languages (LFM), Copenhagen (Denmark)*, volume 70 (2) of *ENTCS*. Elsevier, July 2002.

[9] D. Dietrich. The task-layer of the ΩMEGA system. Diploma thesis, FR 6.2 Informatik, Universität des Saarlandes, Saarbrücken, Germany, 2006.

[10] Xiaorong Huang. *Human Oriented Proof Presentation: A Reconstructive Approach*. Number 112 in DISKI. Infix, Sankt Augustin, Germany, 1996.

[11] Jörg Siekmann, C. Benzmüller, A. Fiedler, Andreas Meier, and Martin Pollet. Proof development with OMEGA: $\sqrt{2}$ is irrational. In Matthias Baaz and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 9th International Conference, LPAR 2002*, number 2514 in LNAI, pages 367–387. Springer, 2002.

[12] A. Trybulec and H. Blair. Computer assisted reasoning with MIZAR. In A. Joshi, editor, *Proceedings of the 9th Int. Joint Conference on Artifical Intelligence*. M. Kaufmann, 1985.

[13] Joris van der Hoeven. Gnu TEXMACS: A free, structured, wysiwyg and technical text editor. Number 39-40 in Cahiers GUTenberg, May 2001.

[14] Lincoln Wallen. *Automated proof search in non-classical logics: efficient matrix proof methods for modal and intuitionistic logics*. MIT Press series in artificial intelligence, 1990.