Teaching Formal Methods and Discrete Mathematics

Mathieu Jaume[1,2]
1. Sorbonne Universités, UPMC Univ. Paris 06, UMR 7606, LIP6, F-75005, Paris, France
2. CNRS, UMR 7606, LIP6, F-75005, Paris, France
Mathieu.Jaume@lip6.fr

Théo Laurent
Sorbonne Universités, UPMC Univ. Paris 06, F-75005, Paris, France
Theo.Laurent@etu.upmc.fr

Despite significant advancements in the conception of (formal) integrated development environments, applying formal methods in software industry is still perceived as a difficult task. To make the task easier, providing tools that help during the development cycle is essential but we think that education of computer scientists and software engineers is also an important challenge to take up. Indeed, we believe that formal methods courses do not appear sufficiently early in compter science curricula and thus are not widely used and perceived as a valid professional skill. In this paper, we claim that teaching formal methods could be done at the undergraduate level by mixing formal methods and discrete mathematics courses and we illustrate such an approach with a small development within FoCaLiZe. We also believe that this could considerably benefit the learning of discrete mathematics.

1 Introduction

Nowadays, critical systems are evaluated according to some security standards like the Common Criteria [7] or according to safety ones like the EN50128 for railways. To reach their high-level rates, these standards require the use of formal methods in order to ensure that security and safety requirements are effectively satisfied by these systems. Indeed, for large developments, ad hoc approaches have proven to be inadequate to ensure that the delivered software truly satisfies safety and security requirements. In fact, the lack of formalisation often leads to produce systems whose behaviors are not fully and precisely understood and described. Formal methods aim at helping to build systems with high safety and security assurances, and formal integrated development environments (F-IDE) embed a variety of such formal methods to help to specify, to document, to implement, to test, to prove or to analyse critical systems. Of course, such environments often ease (and partially automate) the application of formal methods during the development cycle, but developing (and evaluating) critical systems is still a difficult task that requires advanced technical knowledge and large amounts of time. This is certainly one of the reasons why formal methods are still not sufficiently used in industrial software development.

Developing F-IDE that ease the application of formal methods is still a challenging issue but developping a F-IDE which helps to learn formal methods is also a true challenge. We believe that education is the corner stone to promote the use of formal methods in the software creation process. The formal methods community has not enough focused its attention to the education of computer scientists and software engineers, especially at the undergraduate level. Indeed, many computer science curricula do not contain formal methods courses, or such material is not introduced sufficiently early.

Presently, almost all these curricula include discrete mathematics courses but often in isolation from computer science, leaving students understanding little about why (and how) mathematics applies to computer science and *vice versa*. Moreover, teaching discrete mathematics is still often done in the traditional way, using pen and paper, and many computer science students are rather "math-averse" (they

Dubois C., Giannakopoulou D., Mry D. (Eds): Formal Integrated Development Environment 2014 (F-IDE 2014)
EPTCS 149, 2014, pp. 30–43, doi:10.4204/EPTCS.149.4

are more familiar with ASCII characters than with greek alphabet!), perceive mathematics as a difficult discipline and don't understand its relevance in their curricula.

To address this issue, some discrete mathematics courses use functional programming languages (such as ML, OCAML, HASKELL, etc.) to reinforce mathematical concepts. There exists now some discrete mathematics textbooks [9, 17, 26] based on such an approach whose benefits are discussed in [27, 24, 23, 12, 25, 18]. In [25], the author goes further by considering that computer science is also a vital topic for contemporary mathematics students and that they will need some level of competency in programming at some point in their professional practice. Hence, the author claims that the integrated work of mathematics and computer science educators could considerably improve the learning of both subjects: putting functional programming and discrete mathematics in the same course provides a useful service for both computer science and mathematics students. In fact, functional programming languages are high-level languages and thus are well suited to teach discrete mathematics. Indeed, they permit to implement mathematical concepts without considering low level issues such as data representation and memory allocation. Hence, mathematical notions can be easily introduced together with their implementations (that remain very close to the concepts that get implemented) and can be manipulated by students. This is a true way to reinforce their understanding of mathematical concepts. The benefit is also great on the programming side. Using a programming language to learn mathematical concepts leads to handle these concepts as a specification for the program under development and introduce students to the formal specification world. Then, reasoning on the specification and the associated program is a way to smoothly introduce the students to induction, logics and semantics, all notions needed to demonstrate that a program meets its specification. The goal on the computer science side is to put the emphasis on the correctness of the computation which is one of the main purposes of formal methods. Currently, when they are used, programming languages only serve as a formalism to manipulate the computational part of mathematical objects but not to express specifications or to implement proofs. This may lead students to view formal methods as *a posteriori* methods in the programming tasks. In this paper, we claim that formal methods also provide an *a priori* help during the conception of software that can be taught in discrete mathematics courses: specifying a hierarchy of mathematical discrete structures is a good introduction to the design of software architecture.

Even if proof assistants seem now to be mature enough to be adaped to the education, at undergraduate level, formal reasoning is seldom introduced and mostly appears in "pure" logic courses. For example, in [13], the design of a web interface for Coq used to teach logic to undergraduate students is presented. In the context of computer science teaching, formal reasoning is generally introduced at a more advanced stage. This can be done by implementing some automated theorem proving techniques (like in [11]) or by using proof assistants such as Coq or Isabelle. In this case, F-IDE and theorem proving are not objects of the study but are rather considered as a framework for teaching something else. Hopefully, using a language as a vehicle for reinforcing concepts inevitably leads to learn some methodological and practical knowledges about it. For example, [16] is a semantics textbook (to master students) which is entirely based on the proof assistant Isabelle. The main benefit of using a proof assistant in the teaching of semantics is that it allows students to experiment their specifications and to make proofs by using a computer program, which guides them through the development of a completely correct proof and gives them immediate feedback. This avoids students to produce "almost-but-not-completely-right proofs" (as called by Pierce in [19]) or even worse "LSD trip proofs" (as called by Nipkow in [15]).

As we said, we think that teaching formal methods to beginners is essential to disseminate their use in the software industry. However, at the undergraduate level, no prerequesites on computer science can be assumed and we can only suppose some very basic knowledges in mathematics that are also considered as prerequisites for the first courses of discrete mathematics. Hence, we believe that using a F-IDE could be helpful to teach both computer science and discrete mathematics in a mixed course.

This paper aims at presenting our pedagogical approach of both disciplines through a small mathematical development. In this context, the F-IDE used as a teaching tool must be suitable to express specifications (i.e. properties), to write programs (i.e. definitions) and to make proofs. One of the main issues is concerned with proofs. Within most theorem provers, proofs are sequences of commands (belonging to a scripting language) that are hard to read for the human: they lack the information what is being proved at each point, and they lack structure. Such provers are clearly not suitable to teach discrete mathematics at the undergraduate level since they do not provide a proof language close to the informal language of mathematics. Furthermore, the proof language used must be abstract enough to avoid to teach the fine structure of logic (the inference rules) and to automate the "trivial" steps of proofs by allowing students to only express what intermediate steps might help the proof assistant to complete proofs. For these reasons, we think that the FoCaLiZe [10] F-IDE is a good candidate to teach both computer science and discrete mathematics at the undergraduate level. Indeed, FoCaLiZe is an object-oriented programming environment that combines specifications, programs and proofs in the same language, and permits declarative proof descriptions inspired by Lamport's work [14, 6]. These features can be used to formally express specifications and to develop the design and implementation of software as well as some hierarchy of mathematical structures, while proving that implementations (i.e. definitions) meet their specifications or design requirements (i.e. the properties that they are supposed to satisfy). Moreover, the object-oriented features of this language enable the development of an implementation by iterative refinement of its specification: many software components implemented can be built by inheritance and parameterization from already defined components.

2 From binary relations to functions

In this section, we present a small development illustrating how FoCaLiZe can be used to teach basic notions on binary relations and functions and how, at the same time, some knowledge on F-IDE usage can be introduced. To validate our approach we simultaneously introduce concepts involved in FoCaLiZe and discrete mathematics.

Specification of binary relations In FoCaLiZe, the primitive entity of a development is the *species*. Species are the nodes of the hierarchy of structures that makes up a development. A species can be seen as a set of "things", called methods, related to the same concept. As in most modular design systems (i.e. object-oriented, abstract data types, etc.) the idea is to group a data structure with the operations on the data structure, the specification of these operations (in the form of properties), the representation requirements, and the proofs of the properties. Therefore there are three kinds of methods: the carrier type, the programming methods which are functions and the logical methods which are statements, called here properties, and proofs. Each method is identified by its name and can be either declared (primitive constants, operations and properties) or defined (implementation of operations, proofs of theorems).

In discrete mathematics, objects are often defined at an abstract level. For example, a binary relation R is generally defined as a subset of a cartesian product $A \times B$. In fact, to define a relation we first need two sets A and B from which the relation can be built: we don't know anything about these sets but we have to be able to manipulate their elements to describe elements belonging to R. Hence, the species `Binary_relations` of binary relations is parameterized by the sets A:`Setoid` and B:`Setoid` (where the species `Setoid` specifies non-empty sets together with an equivalence relation `equal`, see table 1).

Indeed, an important feature of FoCaLiZe is the ability to parameterize a species by generic collections instanciating a species. Such a mechanism allows us to use a species, without embedding its methods (which is the inheritance mechanism) in the new structure but to use it as a tool box to build this new structure by calling its methods explicitly without knowing how the methods — the tools — are built.

Each species must have one unique carrier method, or representation type: it corresponds to the concrete representation of the elements of the set underlying the structure defined by the species. The carrier is represented by the keyword `Self` inside the species and outside, by the name of the species itself, so that we identify the set with the structure, as usual in mathematics. Like all the other methods, the carrier can be either declared or defined. A declared carrier denotes any set (as in the sentence "let E be a set"), while a defined one is a binding to a concrete type.

In the species `Binary_relations`, nothing is said about how to implement relations and the carrier method `Self` is only declared: we write `R:Self` to express that R is a relation belonging to the species `Binary_relations`. In this context, we are now in position to specify what is a binary relation by introducing a method `relation: Self -> A -> B -> bool` corresponding to characteristic functions of relations (given a relation `R:Self`, for $a : A$ and $b : B$, `relation(R,a,b)=true` iff $(a,b) \in R$). At this level of the hierarchy, the method `relation` is only declared (we don't describe particular relations but only what is needed to define a relation).

Another important feature of FoCaLiZe is the inheritance mechanism: one can enrich a species with additional operations (methods) and redefine some methods of the parent species, but one can also get closer to a runnable implementation by providing explicit definitions to methods that were only declared in the parents. A species can inherit the declarations and definitions of one or several already defined species and is free to define or redefine an inherited method as long as such (re)definition does not change the type of the method.

For example, in mathematics, the set of binary relations is endowed with a notion of equality derived from the equalities of the two component sets. This equality turns this set of binary relations into a setoid. We can easily express that point by indicating that the species `Binary_relations` inherits from the species `Setoid`. In this way, in `Binary_relations` and in all species inheriting from it, the method `equal` can be called to compare relations. Moreover, since the parameters A and B are also setoids, the syntactic construction `A!equal` (resp. `B!equal`) can be used to call the method `equal` of the species A (resp. B) to compare elements of A (resp. of B).

```
species Binary_relations (A is Setoid, B is Setoid) =
inherit Setoid;
signature relation :  Self -> A -> B -> bool;
end ;;
```

Of course, (we hope that) many students know what is a binary relation. However, here, introducing the species of binary relations leads to introduce (at a very basic level) computer science concepts such as parameters, inheritance, abstract and concrete data types, declarations and definitions.

Specifications, Definitions and Proofs At this point, the method `equal` is only declared in the species `Setoid` and it remains to define it in the species `Binary_relations`. To achieve this goal, we can declare the method `is_contained : Self -> Self -> bool` such that `is_contained(R1,R2)=true` iff $R_1 \subseteq R_2$. Hence, we add the signature of `is_contained` together with a property expressing the specification of this method in the species `Binary_relations`.

```
signature is_contained:  Self -> Self -> bool ;
property is_contained_spec:  all r1 r2:  Self,
  is_contained(r1, r2) <-> all a:  A, all b:  B, relation(r1, a, b) -> relation(r2, a, b);
```

Declared methods are introduced by the keyword `signature` while defined methods are introduced by `let` and recursive definitions must be explicitely flagged with the keyword `rec`. The method `is_contained_spec` corresponds to a logical method. Such methods represent the properties of programming methods. The declaration of a logical method is simply the statement of a property, while the definition is a proof of this statement. In the first case, we speak of properties (`property`) that are still to be proved later in the development, while in the second case we speak of theorems (`theorem`). The language also permits logical definitions (`logical let`) to bind names to logical statements. The language used for the statements is composed of the basic logical connectors `\/`, `/\`, `->`, `<->`, `not`, and universal (`all`) and existential (`ex`) quantification over a FoCaLiZe type.

As we can see in our example, as usual during a formal development (and as required as a good practice when applying formal methods), specifications are provided before implementations. Later, during inheritance, the method `is_contained` will have to be implemented and the proof of `is_contained_spec` will have to be done. However, even if this method is only declared, it is possible to use it in a definition. For example, we can now define the method `equal` (which is still only declared) over relations and we can prove the required properties on this definition (as specified in the species `Setoid`, this method must define a reflexive, symmetric and transitive relation over `Self`).

```
let equal(x, y) = is_contained(x, y) && is_contained(y, x) ;
theorem equal_spec :  all r1 r2 :  Self,
  equal (r1, r2) <-> (all a :  A, all b :  B, relation(r1, a, b) <-> relation (r2, a, b))
  proof = by definition of equal
             property is_contained_spec ;
proof of equal_reflexive = by property equal_spec;
proof of equal_symmetric = by property equal_spec;
proof of equal_transitive = by property equal_spec;
```

In fact, the method `equal` is defined together with a proved theorem `equal_spec` corresponding to its specification. The proof is obtained in an automatic way: we just specify here that it can be done by considering the definition of `equal` and the specification `is_contained_spec` (we don't specify how these methods have to be used to make the proof). Thanks to this theorem, proofs of reflexivity, symmetry and transitivity of `equal` are obvious and can also be automatically done (it suffices to indicate that they can be obtained by considering the theorem `equal_spec`).

There are no difficulties to do such mathematical proofs, which can be more detailed if needed to point out the mathematical reasonment. Now, there is, on the computer science side, a question which naturally arises from this tiny development. What is the consequence of redefining the equality in a species inheriting from `Binary_relations`? Any proof relying on the definition of `equal` should be redone (and the compiler leaves no room to an attempt to keep the old version). This is the time to try another version by directly using the definition of `equal` to prove reflexivity, symmetry and transitivity and to find out that these proofs have to be invalidated when redefining `equal`. This puts the emphasis on the benefit obtained from the introduction of the specification of `equal`: only the proof of `equal_spec` is to be redone in case of redefinition of `equal` while the proofs of reflexivity, symmetry and transitivity remain valid since they do not depend on the definition of `equal`. Hence, it is demonstrated that, to minimize the impact of redefinitions, proofs must rely on specification properties instead on definitions (this point is discussed in [21]).

Therefore, as we can see here, even in a very simple and small example on discrete mathematics, some non-trivial methodological issues in computer science can be addressed.

Formal reasoning on mathematical properties At an abstract level, FoCaLiZe allows us to introduce some properties. For example, in the context of a discrete mathematics course, one can define what is

```
species Setoid =
 inherit Basic_object;
 signature element : Self;
 signature equal : Self -> Self -> bool;
 property equal_reflexive : all x : Self, equal(x,x);
 property equal_symmetric : all x y : Self, equal(x,y) -> equal(y,x);
 property equal_transitive : all x y z : Self, equal(x,y) -> equal(y,z) -> equal(x,z);}
 let different (x, y) = not (equal(x,y));
 theorem same_is_not_different : all x y : Self, different(x,y) <-> not (equal(x,y))
  proof = by definition of different;
end;;
```

Table 1: Species of setoids

an injective relation, a surjective relation, a deterministic relation and a left-total relation by adding the following methods in the species `Binary_relations`.

```
logical final let is_left_unique(r) = all a1 a2 : A, all b : B,
            (relation(r,a1,b) /\ relation(r,a2,b)) -> A!equal(a1,a2);
logical final let is_right_total(r) = all b : B, ex a : A, relation(r,a,b);
logical final let is_right_unique(r) = all a : A, all b1 b2 : B,
            (relation(r,a,b1) /\ relation(r,a,b2)) -> B!equal(b1,b2);
logical final let is_left_total(r) = all a : A, ex b : B, relation(r, a, b);
```

These methods correspond to definitions of logical properties: they only bind names to statements and don't intend to express that these properties are true or false (contrarily to the methods introduced by `property`). The keyword `final` is used to forbid the redefinition of these methods in the species inheriting from `Binary_relations`. We can also describe the empty relation, the full relation, and singleton relations as follows.

```
logical final let is_empty_r(r) = all a : A, all b : B, not relation(r,a,b) ;
logical final let is_full_r(r) = all a : A, all b : B, relation(r,a,b) ;
logical final let is_singleton_r(r,a,b) = all a1 : A, all b1 : B,
            relation(r,a1,b1) <-> (A!equal(a,a1) /\ B!equal(b,b1));
```

Similarly, we can introduce operations by only specifying their properties (like in logic programming languages). For example, we can describe union, intersection and difference of relations as follows.

```
logical final let is_union_r(r1,r2,r3) = all a : A, all b : B,
            relation(r3,a,b) <-> (relation(r1,a,b) \/ relation(r2,a,b));
logical final let is_intersection_r(r1,r2,r3) = all a : A, all b : B,
            relation(r3,a,b) <-> (relation(r1,a,b) /\ relation(r2,a,b));
logical final let is_diff_r(r1,r2,r3) = all a : A, all b : B,
            relation(r3,a,b) <-> (relation(r1,a,b) /\ not relation(r2,a,b));
```

Thanks to these methods, it becomes possible to prove classical properties, often done as exercices during discrete mathematics courses. For example we can prove the following property.

$$\left(\begin{array}{l} R_1 \text{ is injective} \wedge R_2 \text{ is injective} \\ \wedge \forall a_1, a_2 : A\, \forall b : B\, ((a_1, b) \in R_1 \wedge (a_2, b) \in R_2) \Rightarrow a_1 = a_2 \end{array}\right) \Leftrightarrow R_1 \cup R_2 \text{ is injective}$$

In the context of a discrete mathematics course, the goal is not here to make the proofs with the automatic features of Zenon but to write a detailed proof of a mathematical property. Hence, we would like to formally prove the following theorem.

```
theorem union_is_left_unique :  all r1 r2 r3 :  Self,
  is_union_r(r1,r2,r3) ->
   ((is_left_unique(r1) /\ is_left_unique(r2)
   /\ (all a1 a2: A, all b: B,((relation (r1,a1,b) /\ relation(r2,a2,b)) -> A!equal(a1,a2))))
   <-> is_left_unique(r3))
```

Within FoCaLiZe, a proof is a tree where the programmer introduces names (`assume`) and hypotheses (`hypothesis`), gives a statement to prove (`prove`) and then provides justification for the statement. This justification can be: (1) a "`conclude`" clause for fully automatic proof; (2) a "`by`" clause with a list of definitions, properties, hypotheses, previous theorems, and previous steps (subject to some scoping conditions) for use by the automatic prover; (3) a sequence of proofs (with their own assumptions, statements, and proofs) whose statements will be used by the automatic prover to prove the current statement. Hence, each step of a proof is independent of the others and can be reused in a similar context[1]. Thanks to these features, as illustrated in table 2, a formal proof (left side of table 2), very close to the informal proof (right side of table 2), of the theorem `union_is_left_unique` can be done within FoCaLiZe. As we can see, the structure of the proof appears clearly (proving an equivalence leads to prove two implications, proving an implication leads to assume hypothesis and to prove the conclusion, proving a conjunction leads to prove each member of the conjunction, using an implication to prove a statement leads to prove hypothesis of this implication, etc.) and each step is clearly characterized by some assumptions and a goal to prove. Hence, using FoCaLiZe during a mathematics course can guide students when specifying and proving classical properties by providing some help to answer questions: is this specification correct according to this property ? are these properties needed to prove this statement ? is there an implicit assumption in this proof ? is this statement provable by using these proof steps ? etc.

Building a hierarchy of mathematical structures Adding the specifications of operations over relations and the classical properties over relations in the species `Binary_relations` only leads to bind names to properties without asserting if these properties are true or false. It is now possible to build a hierarchy of species inheriting from `Binary_relations` in order to constrain relations to satisfy some of these properties. For example, the species of injective relations can be introduced as follows (we just consider here one theorem to illustrate exercices that can be done at this level).

```
species Injective_relations(A is Setoid, B is Setoid) =
 inherit Binary_relations(A, B);
 property left_unique : all r : Self, is_left_unique(r);
 theorem injective_union : all r1 r2 r3 : Self,
    is_union_r(r1,r2,r3)
    -> (all a1 a2: A, all b: B,((relation(r1,a1,b) /\ relation(r2,a2,b))->A!equal(a1,a2)))
  proof = by property left_unique, union_is_left_unique ;
end;;
```

Here, we can use the theorem `union_is_left_unique` and the property `left_unique` (necessarily satisfied by all elements of type `Self`) to prove properties over union of relations (note that the definition of `is_left_unique` is not used in this proof which is obtained by only considering properties of logical connectors between statements). This can also be done for all the operations and properties previously introduced. For example, we can introduce the species of deterministic and left-total relations as follows.

```
species Deterministic_relations (A is Setoid, B is Setoid) =
 inherit Binary_relations (A, B);
  property right_unique : all r : Self, is_right_unique (r);
end;;
species Left_total_relations (A is Setoid, B is Setoid) =
 inherit Binary_relations (A, B);
 property left_total : all r : Self, is_left_total (r);
end;;
```

Furthermore, we can go one step further and build a "complete" hierarchy by considering functions, injective functions, surjective functions and bijective functions as particular cases of relations. This leads to build the following hierarchy of relations corresponding to usual contents in a mathematics course.

[1] This eases maintenance of proofs, and allows us to use exactly the same proof for a statement based on an hypothesis *A* and for the same statement based on a stronger hypothesis *B*, provided the automatic prover can make the inference from *B* to *A*.

```
proof =
  <0>1 assume r1 r2 r3 : Self,
        hypothesis Hunion : is_union_r(r1,r2,r3),
        prove (is_left_unique(r1) /\ is_left_unique(r2)
                 /\(all a1 a2 : A, all b : B,
                     ((relation(r1,a1,b) /\ relation(r2,a2,b))
                      -> A!equal (a1, a2))))
              <-> is_left_unique(r3)

    <1>1 hypothesis Hlu1 : is_left_unique(r1),
           hypothesis Hlu2 : is_left_unique(r2),
           hypothesis Heq : all a1 a2 : A, all b : B,
                 ((relation (r1,a1,b) /\ relation(r2,a2,b))
                 -> A!equal(a1,a2)),
           prove is_left_unique(r3)
      <2>1 assume a1 a2 : A, assume b : B,
             hypothesis Ha1 : relation(r3,a1,b),
             hypothesis Ha2 : relation(r3,a2,b),
             prove A!equal(a1, a2)

        <3>1 hypothesis H11 : relation(r1,a1,b),
               hypothesis H12 : relation(r1,a2,b),
               prove A!equal(a1,a2)
               by hypothesis H11, H12, Hlu1
                  definition of is_left_unique
        <3>2 hypothesis H21 : relation(r2,a1,b),
               hypothesis H22 : relation(r2,a2,b),
               prove A!equal(a1,a2)
               by hypothesis H21, H22, Hlu2
                  definition of is_left_unique
        <3>3 hypothesis H31 : relation(r1,a1,b),
               hypothesis H32 : relation(r2,a2,b),
               prove A!equal(a1,a2)
               by hypothesis H31, H32, Heq
        <3>4 hypothesis H41 : relation(r2,a1,b),
               hypothesis H42 : relation(r1,a2,b),
               prove A!equal(a1,a2)
               by hypothesis H41, H42, Heq
        <3>f qed by step <3>1, <3>2, <3>3, <3>4
                     hypothesis Hunion,
                                Ha1, Ha2
                     definition of is_union_r

    <1>2 hypothesis Hlu3 : is_left_unique(r3),
          prove is_left_unique(r1) /\ is_left_unique(r2)
          /\ (all a1 a2 : A, all b : B,
          ((relation(r1,a1,b)/\relation(r2,a2,b))->A!equal(a1,a2)))
      <2>1 prove is_left_unique(r1)
        <3>1 assume a1 a2 : A, assume b : B,
               hypothesis Hr1:relation(r1,a1,b)/\relation(r1,a2,b),
               prove A!equal(a1,a2)
          <4>1 prove relation(r3,a1,b) /\ relation(r3,a2,b)
                 by hypothesis Hr1, Hunion definition of is_union_r
          <4>f qed by step <4>1 hypothesis Hlu3
                       definition of is_left_unique
        <3>f qed by step <3>1 definition of is_left_unique
      <2>2 prove is_left_unique(r2)
        <3>1 assume a1 a2 : A, assume b : B,
               hypothesis Hr2:relation(r2,a1,b)/\relation(r2,a2,b),
               prove A!equal(a1,a2)
          <4>1 prove relation(r3,a1,b) /\ relation(r3,a2,b)
                 by hypothesis Hr2, Hunion definition of is_union_r
          <4>f qed by step <4>1 hypothesis Hlu3
                       definition of is_left_unique
        <3>f qed by step <3>1 definition of is_left_unique
      <2>3 prove all a1 a2 : A, all b : B,
               ((relation(r1,a1,b) /\ relation(r2,a2,b))
               -> A!equal(a1,a2))
        <3>1 assume a1 a2 : A, assume b : B,
              hypothesis H0:relation(r1,a1,b)/\relation(r2,a2,b),
              prove relation(r3,a1,b) /\ relation(r3,a2,b)
              by hypothesis H0, Hunion definition of is_union_r
        <3>f qed by step <3>1 hypothesis Hlu3
                     definition of is_left_unique
      <2>f conclude
    <1>f conclude

  <0>f conclude;
```

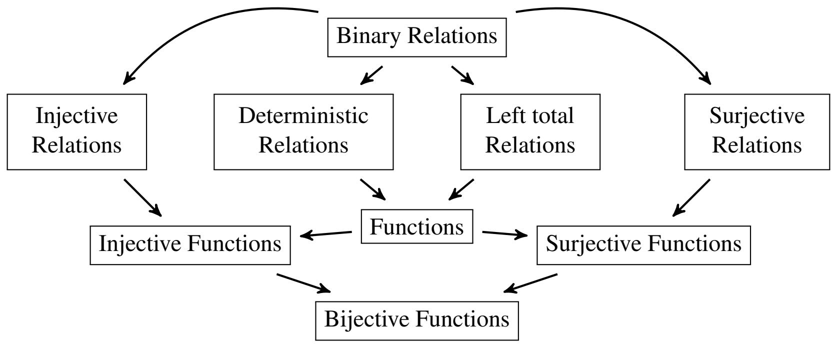
Let R_1, R_2 and R_3 be binary relations.
such that $R_3 = R_1 \cup R_2$.
Let us prove the desired equivalence.

First, let us suppose that R_1 is injective,
R_2 is injective,
and that $\forall a_1, a_2 : A \forall b : B$
$((a_1, b) \in R_1 \wedge (a_2, b) \in R_2) \Rightarrow a_1 = a_2$
and let us prove that R_3 is injective.
Let $a_1, a_2 : A$ and $b : B$ be elements
such that $(a_1, b) \in R_3$
and $(a_2, b) \in R_3$,
and let us prove that $a_1 = a_2$.
We consider 4 cases.
If we suppose that $(a_1, b) \in R_1$,
and $(a_2, b) \in R_1$,
then we can prove $a_1 = a_2$
since (by hypothesis) R_1 is injective,
and by definition of an injective relation.
If we suppose that $(a_1, b) \in R_2$,
and $(a_2, b) \in R_2$,
then we can prove $a_1 = a_2$
since (by hypothesis) R_2 is injective,
and by definition of an injective relation.
If we suppose that $(a_1, b) \in R_1$,
and $(a_2, b) \in R_2$,
then we can prove $a_1 = a_2$
by using hypothesis (`Heq`).
If we suppose that $(a_1, b) \in R_2$,
and $(a_2, b) \in R_1$,
then we can prove $a_1 = a_2$
by using hypothesis (`Heq`).
In these 4 cases, $a_1 = a_2$ and
since by hypothesis $R_3 = R_1 \cup R_2$,
and $(a_1, b) \in R_3$ and $(a_2, b) \in R_3$,
we can conclude by definition of $\cup$.

Now, let us suppose that R_3 is injective,
and let us prove that R_1 and R_2 are injective,
and are such that $\forall a_1, a_2 : A \, \forall b : B$,
$((a_1, b) \in R_1 \wedge (a_2, b) \in R_2) \Rightarrow a_1 = a_2$
We first prove that R_1 is injective.
Let $a_1, a_2 : A$ and $b : B$ be elements
such that $(a_1, b) \in R_1 \wedge (a_2, b) \in R_1$,
and let us prove that $a_1 = a_2$.
We prove that $(a_1, b) \in R_3 \wedge (a_2, b) \in R_3$
since $R_3 = R_1 \cup R_2$ and by definition of $\cup$.
Hence, since R_3 is injective, we get $a_1 = a_2$
by definition of an injective relation.
Thus, by definition, R_1 is also injective.
Similarly we prove that R_2 is injective.
Let $a_1, a_2 : A$ and $b : B$ be elements
such that $(a_1, b) \in R_2 \wedge (a_2, b) \in R_2$,
and let us prove that $a_1 = a_2$.
We prove that $(a_1, b) \in R_3 \wedge (a_2, b) \in R_3$
since $R_3 = R_1 \cup R_2$ and by definition of $\cup$.
Hence,since R_3 is injective, we get $a_1 = a_2$
by definition of an injective relation.
Thus, by definition, R_2 is also injective.
It remains to prove that
$\forall a_1, a_2 : A \forall b : B \, ((a_1, b) \in R_1 \wedge (a_2, b) \in R_2)$
$\Rightarrow a_1 = a_2$
Let $a_1, a_2 : A$ and $b : B$ be elements
such that $(a_1, b) \in R_1 \wedge (a_2, b) \in R_2$.
We can prove that $(a_1, b) \in R_3 \wedge (a_2, b) \in R_3$
since $R_3 = R_1 \cup R_2$ and by definition of $\cup$.
Hence,since R_3 is injective, we get $a_1 = a_2$
by definition of an injective relation.
This concludes the proof of the conjunction `<2>1`.
This concludes the proof of the equivalence `<0>1`.

This concludes the proof of the theorem.

Table 2: Proof of theorem `union_is_left_unique`



However, in the species `Functional_relations` of functions (and in all the species inheriting from it), elements of type `Self` are still defined by their characterisitic functions `relation`: this leads to view functions from A to B as particular cases of relations over $A \times B$. However, it may be useful to declare a method `fct : Self -> A -> B` corresponding to the usual concept of functions (known by students at the undergraduate level). From this method, it becomes possible to define the method `relation` and to prove the required properties. This can be easily done as follows.

```
species Functional_relations (A is Setoid, B is Setoid) =
 inherit Left_total_relations(A, B), Deterministic_relations(A, B);
 signature fct : Self -> A -> B;
 let relation(r,x,y) = B!equal(fct(r,x),y);
 proof of right_unique = by definition of relation, is_right_unique
                            property B!equal_symmetric, B!equal_transitive;
 proof of left_total = by definition of relation, is_left_total property B!equal_reflexive;
end;;
```

In addition to the mathematical contents of these specifications (allowing students to understand at a deep level the differences between functions and relations, and the main properties these objects), using FoCaLiZe to describe the hierarchy of relations and functions allows students to consider multiple-inheritance and computational notions.

Implementations and their properties Until now, we have only used FoCaLiZe to express, to prove and to design the architecture of mathematical properties. The next step consists in introducing concrete data types, recursive programming and inductive proofs over mathematical objects. We show here how to introduce these notions by implementing finite parts of a set by lists. We first define the species (parameterized by a setoid `S`) of finite parts of `S` (due to space limitations, we only consider the methods needed in our example, but, of course, this species contains many other methods).

```
species Finite_parts(S is Setoid) =
 inherit Setoid ;
 signature belongs: S -> Self -> bool;
 signature cardinal: Self -> int;
 signature empty : Self;
 signature release : Self -> S -> Self;
 property release_spec : all x : Self, all t1 t2 : S,
    belongs(t1,release(x,t2)) <-> (S!different(t1,t2) /\ belongs(t1,x));
 property empty_spec :all t: S, not belongs(t,empty);
 signature from_list : list(S) -> Self ;
 property belongs_spec: all t: list(S), all h x: S,
  (belongs(x,from_list(t)) \/ S!equal(h,x)) <-> belongs(x,from_list(h::t));
end;;
```

Hence, a finite part `P:Self` of `S` is described by a membership relation (`belongs(s,P)=true` iff $s \in P$) and by its cardinal (which is finite since it is represented here by an integer). In our example, we consider the methods `empty` (for the empty part) and `release` (that permits to remove an element from a finite part). At this abstract level, these methods are only declared together with their specifications. Furthermore, we declare a method `from_list` that aims at building a finite part from elements belonging to a list and which is used to specify the method `belong`. We can now refine this species by representing finite parts with the concrete FoCaLiZe type `list` of lists. Within FoCaLiZe, the language used for the programming methods is similar to the functional core of OCaml (let-binding, pattern matching, conditional, higher order functions, etc), with the addition of a construction to call a method from a given structure. Thanks to these constructions, we can introduce the species `Finite_parts_by_lists` inheriting from `Finite_parts` and providing definitions for the programming methods.

```
species Finite_parts_by_lists(S is Setoid) =
 inherit Finite_parts(S);
 representation = list(S);
 let rec belongs(x:S,l) = match l with | [] -> false
    | h :: q ->  S!equal(h,x) || (belongs(x,q))
 termination proof = structural l;
 let rec cardinal(e) = match e with | [] -> 0
    | _ ::t -> 1 + cardinal(t)
 termination proof = structural e;
 let empty = [];
 let rec release(e,s) = match e with | [] -> []
  | h::t -> if S!equal(s,h) then release(t,s) else h::release(t,s)
  termination proof = structural e ;
 let from_list (s:list(S)):Self = s;
end;;
```

In the context of a discrete mathematics course, this leads to introduce recursive definitions and to (lightly) address termination issues of such definitions.

We are now in position to prove all the properties stated in the species `Finite_parts`. We just present here the proof of `release_spec`. The proof is done by induction on lists, and, here again, as we can see in table 3, the formal proof is very close to the informal one (in the informal proof we write $e \ominus s$ instead of `release(e,s)`): the empty list case and the inductive step are independently proved, and the properties and definitions leading to intermediate results are made explicit, as well as the context in which such results are proved. Each method of the species `Finite_parts_by_lists` is now defined.

Within FoCaLiZe, a collection can be built upon a completely defined species. This means that every method must be defined. In other words, in a collection, every operation has an implementation, and every theorem is formally proved. In addition, a collection is "frozen": it cannot be used as a parent of a species in the inheritance graph. Moreover, to ensure modularity and abstraction, the carrier of a collection is hidden: seen from the outside, it becomes an abstract type. This means that any software component dealing with a collection will only be able to manipulate it through the operations it provides (i.e. its methods). This point is especially important since it prevents other software components from breaking representation invariants required by the internals of the collection.

3 Conclusion

Using a computer to teach discrete mathematics at the undergraduate level is usally done by considering functionnal programming languages allowing students to formally express computational contents of mathematical concepts by programs and to informally reason on these programs. In this paper, we go one step futher by showing that abstract specifications and proofs can also be implemented at this level whitout assuming some advanced theoretical background. Indeed, while teaching experiences using F-IDE are mostly done at master level, we believe that such approaches can also be adopted for beginning

```
proof of release_spec =
<0>1 assume e1 e2 : S,
     prove all l:list(S),belongs(e1,release(l,e2))
     <-> (S!different (e1,e2) /\belongs(e1,l))
 <1>b prove belongs(e1,release([],e2))
      <-> (S!different(e1, e2) /\ belongs(e1,[]))
  <2>1 prove not (belongs(e1, release([], e2)))
       by definition of release, empty
          property empty_spec
  <2>2 prove not (S!different (e1,e2) /\ belongs(e1,[]))
       by definition of empty property empty_spec
  <2>f conclude
 <1>i assume t: list(S), assume h: S,
      hypothesis HI : (belongs(e1, release(t, e2))
        <-> (S!different(e1,e2) /\ belongs(e1,t))),
      prove (belongs(e1,release(h::t,e2))
        <-> (S!different(e1,e2) /\ belongs(e1,h::t)))
  <2>1 hypothesis H1 : belongs(e1,release(h::t,e2)),
       prove S!different(e1,e2) /\ belongs(e1,h::t)
   <3>1 hypothesis C1 : S!equal(e2,h),
        prove S!different(e1,e2) /\ belongs(e1,h::t)
    <4>1 prove S!different(e1,e2)
         by definition of release
            hypothesis H1, C1, HI
    <4>2 prove belongs(e1,h::t)
         <5>1 prove belongs(e1,t)
              by hypothesis HI, C1, H1
                 definition of release
         <5>f qed by step <5>1
                    property belongs_spec
                    definition of from_list
    <4>f conclude
   <3>2 hypothesis C2 : not S!equal(e2,h),
        prove S!different(e1,e2) /\ belongs(e1,h::t)
    <4>1 hypothesis C21 : S!equal(e1,h),
         prove S!different(e1,e2) /\ belongs(e1,h::t)
     <5>1 prove S!different(e1,e2)
          by hypothesis C2, C21
             property S!equal_symmetric, S!equal_transitive,
                      S!same_is_not_different
     <5>2 prove belongs(e1,h::t)
          by property belongs_spec, S!equal_symmetric
             hypothesis C21 definition of from_list
     <5>f conclude
    <4>2 hypothesis C22 : not S!equal(e1,h),
         prove S!different(e1,e2) /\ belongs(e1,h::t)
     <5>1 prove belongs(e1, h::release(t, e2))
          by hypothesis H1, C2
             definition of release
     <5>2 prove belongs(e1,release(t,e2))
          by step <5>1 hypothesis C22
             definition of belongs
             property belongs_spec, S!equal_symmetric
     <5>3 prove S!different(e1,e2) /\ belongs(e1,t)
          by step <5>2 hypothesis HI
     <5>4 prove belongs(e1,h::t)
          by step <5>3 definition of belongs
             property belongs_spec
     <5>f conclude
    <4>f conclude
   <3>f conclude
  <2>2 hypothesis H2: S!different(e1,e2) /\ belongs(e1,h::t),
       prove belongs(e1,release(h::t,e2))
   <3>1 hypothesis C1 : S!equal(e1,h),
        prove belongs(e1,release(h::t,e2))
    <4>1 prove not S!equal(e2,h)
         by hypothesis H2, C1
            property S!equal_transitive, S!equal_symmetric,
                     S!same_is_not_different
    <4>2 prove release(h::t,e2)=h::release(t,e2)
         by step <4>1 definition of release
    <4>3 prove belongs(e1,h::release(t,e2))
         by definition of belongs hypothesis C1
            property  belongs_spec, S!equal_symmetric
    <4>f qed by step <4>2, <4>3
   <3>2 hypothesis C2 : not S!equal(e1,h),
        prove belongs(e1,release(h::t,e2))
    <4>1 prove belongs(e1,t)
         by definition of belongs hypothesis C2, H2
            property belongs_spec, S!equal_symmetric
    <4>2 prove belongs(e1,release(t,e2))
         by step <4>1 hypothesis H2, HI
    <4>3 prove release(h::t,e2) = release(t,e2)
               \/ release(h::t,e2) = h::release(t,e2)
         by definition of release
    <4>f qed by step <4>3, <4>2
            property belongs_spec definition of from_list
   <3>f conclude
  <2>f conclude
 <1>f conclude
<0>f conclude;
```

Let e_1, e_2 be elements of S, and and let us prove (by induction on l) the desired equivalence. First, let us prove the property for the empty list. Since $e_1 \notin [] \ominus e_2 = []$ by definition of $\ominus$ and $[]$ (because $\forall e : S\ e \notin []$) and since $\neg(e_1 \neq e_2 \wedge e_1 \in [])$ (by definition of $[]$ and because $\forall e : S\ e \notin []$) we can conclude.

For the inductive step, let t be a list and $h : S$. By induction hypothesis, we have: $e_1 \in t \ominus e_2 \Leftrightarrow (e_1 \neq e_2 \wedge e_1 \in t)$, and we prove that $e_1 \in h :: t \ominus e_2 \Leftrightarrow (e_1 \neq e_2 \wedge e_1 \in h :: t)$

($\Rightarrow$) Let us suppose that $e_1 \in h :: t \ominus e_2$ (H_1) and let us prove $e_1 \neq e_2 \wedge e_1 \in h :: t$. Two cases are possible and we prove the property for these two cases. If $e_2 = h$,

then we have $e_1 \neq e_2$ since, by definition of $\ominus$, $h :: t \ominus e_2 = t \ominus e_2$, so by H_1, we get $e_1 \in t \ominus e_2$ and by induction hypothesis $e_1 \neq e_2$. In remains to prove $e_1 \in h :: t$. Indeed, we have $e_1 \in t$ since, by definition of $\ominus$, $h :: t \ominus e_2 = t \ominus e_2$, so by H_1, we get $e_1 \in t \ominus e_2$ and by induction hypothesis $e_1 \in t$. Hence, by definition of `from_list` and by property `belongs_spec`, we get $e_1 \in h :: t$. Hence, when $e_2 = h$, we have $e_1 \neq e_2 \wedge e_1 \in h :: t$. Now, let us suppose that $e_2 \neq h$. Two subcases are possible and we prove the property for both cases. If $e_1 = h$,

then $e_1 \neq e_2$ since $e_2 \neq h$ and by properties of equality.

Furthermore we have $e_1 \in h :: t$ (since $e_1 = h$) by symmetry of equality and by definition of `from_list` and property `belongs_spec`.

Now, let us suppose that $e_1 \neq h$.

We have $e_1 \in h :: (t \ominus e_2)$ since $e_2 \neq h$ and $e_1 \in h :: t \ominus e_2$ (H_1) and by definition of $\ominus$, we get $h :: t \ominus e_2 = h :: (t \ominus e_2)$. Furthermore, it follows $e_1 \in t \ominus e_2$ since $e_1 \neq h$ and by definition of the membership relation and by property `belong_spec` and by symmetry of equality. Hence, by induction hypothesis, we get $e_1 \neq e_2 \wedge e_1 \in t$. From $e_1 \in t$, we get $e_1 \in h :: t$ by definition of the membership relation and by property `belong_spec`.

Hence, when $e_2 \neq h$, we also have $e_1 \neq e_2 \wedge e_1 \in h :: t$.

($\Leftarrow$) Let us suppose that $e_1 \neq e_2 \wedge e_1 \in h :: t$ (H_2) and let us prove that $e_1 \in h :: t \ominus e_2$. Two cases are possible. If $e_1 = h$,

then $e_2 \neq h$ by hypothesis H_2 and by properties of equality. Hence it follows $h :: t \ominus e_2 = h :: (t \ominus e_2)$ by definition of $\ominus$. Furthermore, we get $e_1 \in h :: (t \ominus e_2)$ since $e_1 = h$ and by definition of the membership relation and by property `belongs_spec` and symmetry of equality. Hence, when $e_1 = h$, we have $e_1 \in h :: t \ominus e_2$. Now, let us suppose that $e_1 \neq h$.

Then we get $e_1 \in t$ since $e_1 \in h :: t$ and $e_1 \neq h$ and by definition of the membership relation, and by property `belongs_spec` and symmetry of equality. Hence, by induction hypothesis, and since $e_1 \neq e_2$ (H_2) it follows $e_1 \in t \ominus e_2$

Moreover, by definition of $\ominus$ we have $h :: t \ominus e_2 = t \ominus e_2$ or $h :: t \ominus e_2 = h :: (t \ominus e_2)$. Hence, when $e_1 \neq h$, we have $e_1 \in h :: t \ominus e_2$ (by definition of `from_list` and by property `belongs_spec`).

This concludes the inductive step. This concludes the proof by induction. This concludes the proof.

Table 3: Proof of `release_spec`

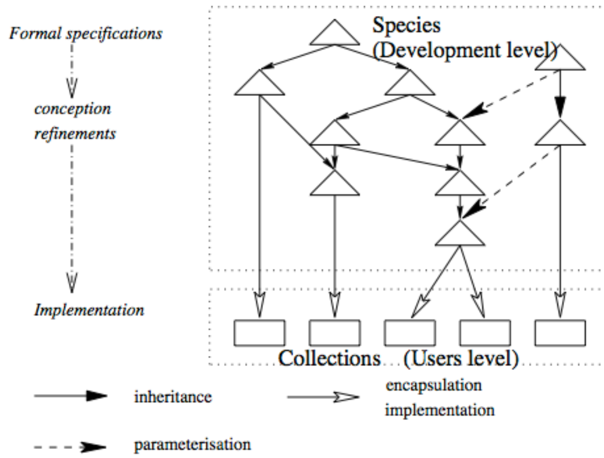


Figure 1: Formal development within FoCaLiZe

students. We claim here that teaching how to develop software with F-IDE to beginners is essential to ease and to promote their use in industry.

FoCaLiZe was conceived from the beginning to help building systems with high safety and security assurances. FoCaLiZe includes a language based on firm theoretical results [20], with a clear semantics and provides an efficient implementation – *via* translation to OCaml. It has functional and object-oriented features and provides means for the programmers to write formal proofs of their code in a more or less detailed way within a declarative proof language based on the Zenon automatic theorem prover [2]. Zenon eases the task of writing formal proofs and translates them into Coq for high-assurance checking. FoCaLiZe also provides powerful features (such as inheritance, parameterization and late-binding) that enable a stepwise refinement methodology to go from specification all the way down to executable code. Indeed, thanks to the main features of FoCaLiZe, a formal development can be organized as a hierarchy (as illustrated in figure 1) which may have several roots: the upper levels of the hierarchy are built during the specification stage while the lower ones correspond to implementations. Thus, FoCaLiZe unifies within the same language the formal modeling work, the development of the code, and the certification proofs. Very important is the ability in FoCaLiZe to have specifications, implementations and proofs within the same language, since it eliminates the errors introduced between layers, at each switch between languages, during the development cycle. Other frameworks, like Atelier B [1], also aims at implementing tools for making formal development a reality. FoCaLiZe doesn't follow the same path, trying to keep the means of expression close to what engineers usually know: a programming language. Of course, nowadays, proof assistants also provide some features for structuring code (module systems, type classes, etc), but most of them still cannot be used to obtain efficient programs. Compilation of FoCaLiZe developments leads to efficient OCaml programs (which are not obtained by extracting computational contents of proofs). It is this focus on efficiency that makes FoCaLiZe a real programming language. To our knowledge, only the Agda [4] programming language, based on dependent types and compiling *via* Haskell, has a comparable mix of features. Note that the FoCaLiZe language is also based on a dependent type language, but with some restrictions on dependencies. Furthermore, FoCaLiZe provides several automatic tools to ease the generation of programs from specifications, the generation of documentation, and the production of test suites [5].

For all these reasons, we think that FoCaLiZe is not only well suited to develop critical systems but is also a good framework to teach both computer science and discrete mathematics courses. For example, we have already used [8] FoCaLiZe (together with Coq) to teach (at a master level) semantics of object-oriented features of programming languages. In this paper, we consider FoCaLiZe as a teaching tool at the undergraduate level and illustrate our approach with a small development introducing very basic concepts of discrete mathematics and showing how to mix both formal methods and discrete mathematics courses. Indeed, FoCaLiZe provides an environment simple enough to be usable by most students at university (even if they are not fully acquainted with theoretical concepts such as higher-order logics), in particular by making development of correct proofs as easy as possible and as readable as possible. Moreover, FoCaLiZe leads to stress the process of abstraction through the construction, step by step, of problem solutions from their specifications. This can be helpful to improve the learning process of discrete mathematics but also to show to students that computer science involves a lot of mathematical activities and *vice versa*.

In addition to pedagogical benefits, we believe that teaching how to use F-IDE as early as possible leads to raise the level of mathematical rigor for computer science so as to ensure that formal methods are perceived as valid professional disciplines by students. Formal methods will be of increasing value in computer and software engineering (especially for safety-critical, security-sensitive, and embedded systems) and we think that education is one challenge to take up in order to promote the dissemination of formal methods in software industry. FoCaLiZe includes a computer algebra library, mostly developed by R. Rioboo [3, 22], which implements mathematical structures up to multivariate polynomial rings and includes complex algorithms with performance comparable to the best computer algebra systems in existence. Hence, as future works, we believe that FoCaLiZe could be used to develop a complete discrete mathematics course.

Acknowledgments The authors would like to thank Renaud Rioboo for his help and for enlightening discussions about how to program with FoCaLiZe and about teaching discrete mathematics.

References

[1] J.R. Abrial (1996): *The B Book: Assigning Programs to Meanings*. Cambridge University Press, doi:10.1017/CBO9780511624162.

[2] R. Bonichon, D. Delahaye & D. Doligez (2007): *Zenon: An Extensible Automated Theorem Prover Producing Checkable Proofs*. In: *Logic for Programming, Artificial Intelligence, and Reasoning, 14th Int. Conf., LPAR*, *LNCS* 4790, Springer, pp. 151–165, doi:10.1007/978-3-540-75560-9_13.

[3] S. Boulmé, T. Hardin & R. Rioboo (2001): *Some hints for polynomials in the Foc project*. In: *9th Symp. on the Integration of Symbolic Computation and Mechanized Reasoning, Calculemus 2001*.

[4] A. Bove, P. Dybjer & U. Norell (2009): *A Brief Overview of Agda - A Functional Language with Dependent Types*. In: *Theorem Proving in Higher Order Logics, 22nd Int. Conf., TPHOLs 2009, Proceedings*, *LNCS* 5674, Springer, pp. 73–78, doi:10.1007/978-3-642-03359-9_6.

[5] M. Carlier, C. Dubois & A. Gotlieb (2010): *Constraint Reasoning in FocalTest*. In: *ICSOFT 2010 - Proceedings of the Fifth Int. Conf. on Software and Data Technologies, Volume 2*, SciTePress, pp. 82–91.

[6] K. Chaudhuri, D. Doligez, L. Lamport & S. Merz (2008): *A TLA$^+$ Proof System*. In: *Proc. of the LPAR Workshops: Knowledge Exchange: Automated Provers and Proof Assistants, and The 7th Int. Workshop on the Implementation of Logics (KEAPPA)*.

[7] Common Criteria (2005): *Common Criteria for Information Technology Security Evaluation, Norme ISO 15408 – Version 3.0 Rev 2*.

[8] D. Delahaye, M. Jaume & V. Prevosto (2005): *Coq: un outil pour l'enseignement*. *Technique et Science Informatiques (TSI)* 24(9), pp. 1139–1160, doi:10.3166/tsi.24.1139-1160.

[9] K. Doets & J. van Eijck (2004): *The Haskell Road to Logic, Maths and Programming*. King's College Publications, London.

[10] Focalize (2010): *Focalize, Tutorial and reference manual*. Distribution available at: `http://focalize.inria.fr`.

[11] J. Harrison (2009): *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, doi:10.1007/s10817-012-9251-8.

[12] P. B. Henderson (2002): *Functional and declarative languages for learning discrete mathematics*. In Proceedings of the Int. Workshop on Functional & Declarative Programming in Education (FDPE 2002), editors: *Published as Technical Report No. 0210 of the University of Kiel (Germany)*.

[13] M. Hendriks, C. Kaliszyk, F. van Raamsdonk & F. Wiedijk (2010): *Teaching logic using a state-of-the-art proof assistant*. *Acta Didactica Napocensia* 3(2), pp. 35–48.

[14] L. Lamport (1995): *How to Write a Proof*. *AMM: The American Mathematical Monthly* 102(7), pp. 600–608, doi:10.2307/2974556.

[15] T. Nipkow (2012): *Teaching Semantics with a Proof Assistant: No more LSD Trip Proofs*. In: *Verification, Model Checking, and Abstract Interpretation (VMCAI 2012)*, *LNCS* 7148, Springer, pp. 24–38, doi:10.1007/978-3-642-27940-9_3.

[16] T. Nipkow & G. Klein (2013): *Concrete Semantics. A proof assistant approach*. Draft. Available at `http://www21.in.tum.de/~nipkow/Concrete-Semantics/concrete_semantics.pdf`.

[17] J. T. O'Donnell, C. V. Hall & R. Page (2006): *Discrete mathematics using a computer*. Springer, doi:10.1007/1-84628-598-4.

[18] R.L. Page (2003): *Software is discrete mathematics*. In: *Proc. of the Eighth ACM SIGPLAN Int. Conf. on Functional Programming, ICFP 2003*, ACM, pp. 79–86, doi:10.1145/944746.944713.

[19] B. C. Pierce (2009): *Lambda, the ultimate TA: using a proof assistant to teach programming language foundations*. In: *Proc. of the 14th ACM SIGPLAN Int. Conf. on Functional programming, ICFP 2009*, ACM, pp. 121–122, doi:10.1007/978-3-642-27940-9_3.

[20] V. Prevosto & D. Doligez (2002): *Algorithms and Proof Inheritance in the Foc language*. *Journal of Automated Reasoning* 29(3-4), pp. 337–363, doi:10.1023/A:1021979218446.

[21] V. Prevosto & M. Jaume (2003): *Making proofs in a hierarchy of mathematical structures*. In: *11th Symp. on the Integration of Symbolic Computation and Mechanized Reasoning, Calculemus 2003*, Aracne, pp. 89–100.

[22] R. Rioboo (2009): *Invariants for the FoCaL language*. *Annals of Mathematics and Artificial Intelligence* 56(3-4), pp. 273–296, doi:10.1007/s10472-009-9156-3.

[23] S. Da Rosa (2002): *The Role of Discrete Mathematics and Programming in Education*. In Proceedings of the Int. Workshop on Functional & Declarative Programming in Education (FDPE 2002), editors: *Published as Technical Report No. 0210 of the University of Kiel (Germany)*.

[24] C. Scharff & A. Wildenberg (2002): *Teaching Discrete Structures with SML*. In Proceedings of the Int. Workshop on Functional & Declarative Programming in Education (FDPE 2002), editors: *Published as Technical Report No. 0210 of the University of Kiel (Germany)*.

[25] T. VanDrunen (2011): *The case for teaching functional programming in discrete math*. In C. Videira Lopes & K. Fisher, editors: *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011*, ACM, pp. 81–86, doi:10.1145/2048147.2048180.

[26] T. VanDrunen (2012): *Discrete mathematics and Functional Programming*. Franklin, Beedle and Associates.

[27] R. L. Wainwright (1992): *Introducing functional programming in discrete mathematics*. In N. B. Dale, editor: *Proc. of the 23rd SIGCSE Technical Symp. on Computer Science Education*, ACM, pp. 147–152, doi:10.1145/134510.134540.