TYPES Summer School 2005

Proofs of Programs and Formalisation of Mathematics

Lecture Notes

Volume II

August 15–26 2005

Göteborg, Sweden

Proofs of Programs and Formalisation of Mathematics

Lecture Notes: Volume II

Introduction to Systems

Advanced Applications and Tools

Dependently Typed Programming

Formalisation of Mathematics

TYPES Summer School

August 15–26 2005

Göteborg, Sweden

Contents

PART III:

Introduction to Systems

Yves Bertot, Jean-Christophe Filliatre:

Coq

Coq in a Hurry

Yves Bertot

June 30, 2005

These notes provide a quick introduction to the Coq system and show how it can be used to define logical concepts and functions and reason about them. It is designed as a tutorial, so that readers can quickly start their own experiments, learning only a few of the capabilities of the system. A much more comprehensive study is provided in [1], which also provides an extensive collection of exercises to train on.

1 Expressions and logical formulas

The Coq system provides a language in which one handles formulas, verify that they are well-formed, and prove them. Formulas may also contain functions and limited forms of computations are provided for these functions.

The first thing you need to know is how you can check whether a formula is well-formed. The command is called `Check`. Here are a few examples, which use a few of the basic objects and types of the system.

```
Check True.
True : Prop

Check False.
False : Prop

Check 3.
3 : nat

Check (3+4).
3 + 4 : nat

Check (3=5).
3=5 : Prop

Check (3,4).
(3,4) : nat * nat
```

```
Check ((3=5)/\True).
3 = 5 /\ True

Check nat -> Prop.
nat -> Prop : Type

Check (3 <= 6).
3 <= 6 : Prop
```

The notation $A : B$ is uniformly used to indicate that the type of the expression A is the expression B.

Among these formulas, some can be read as propositions (they have type `Prop`), others may be read as numbers (they have type `nat`), others may be read as elements of more complex data structures. You can also try to check badly formed formulas and in this case the Coq system returns an informative error statement.

Complex formulas can be constructed by combining propositions with logical connectives, or other expressions with addition, multiplication, the pairing construct, and so on. You can also construct a new function by using the keyword `fun`, which replaces the λ symbol of lambda calculus and similar theories.

```
Check (fun x:nat => x = 3).
fun x : nat => x = 3 : nat -> Prop

Check (forall x:nat, x < 3 \/ (exists y:nat, x = y + 3)).
forall x : nat, x < 3 \/ (exists y : nat, x = y + 3) : Prop

Check (let f := fun x => (x * 3,x) in f 3).
let f := fun x : nat => (x * 3, x) in f 3 : nat * nat
```

Please note that some notations are *overloaded*. For instance, the `*` sign is used both to represent conventional multiplication on numbers and the cartesian product on types. One can find the function hidden behind a notation by using the `Locate` command.

```
Locate "_ <= _".
Notation             Scope
"x <= y" := le x y   : nat_scope
                       (default interpretation)
```

The conditions for terms to be well-formed have two origins: first, the syntax must be respected (parentheses, keywords, binary operators must have two arguments); second, expressions must respect a type discipline. The `Check` not only checks that expressions are well-formed but it also gives the type of expressions. For instance we can use the `Check` command to verify progressively that some expressions are well-formed.

```
Check True.
True : Prop
```

```
Check False.
False : Prop

Check and.
and : Prop -> Prop -> Prop

Check (and True False).
True /\ False : Prop
```

In the last example, `and` is a function that expects an argument of type `Prop` and returns a function of type `Prop -> Prop`. It can therefore be applied to `True`, which has the right type. But the function we obtain expects another argument of type `Prop` and it can be applied to the argument `False`. The notation

$$a \texttt{ -> } b \texttt{ -> } c$$

actually stands for

$$a \texttt{ -> (}b \texttt{ -> } c\texttt{)}$$

and the notation $f\ a\ b$ actually stands for $(f\ a)\ b$.

The last example also shows that the notation `/\` is an infix notation for the function `and`.

Some constructs of the language have a notion of *bound* variable. Among the examples we have already seen, the `forall` and `exist` logical quantifiers and the `fun` function constructor and the `let .. in` local declaration construct have this characteristic. When constructs have a bound variable, this variable can be used with some type inside some part of the construct called the scope. The type is usually given explicitely, but it may also sometimes be left untold, and the Coq system will infer it.

2 Defining new constants

You can define a new constant by using the keyword `Definition`. Here is an example:

```
Definition example1 := fun x : nat => x*x+2*x+1.
```

An alternative, exactly equivalent, definition could be:

```
Definition example1 (x : nat) := x*x+2*x+1.
```

3 Proving facts

The notation A:B is actually used for several purposes in the Coq system. One of these purposes is to express that A is a proof for the logical formula B. This habit is usually referred to under the name *Curry-Howard Isomorphism*. One can find already existing proofs of facts by using the `Search` command.

```
Search True.
I : True
Search le.
le_n : forall n : nat, n <= n
le_S : forall n m : nat, le n m -> le n (S m)
```

The theorem `le_S` uses a function `S`, this function maps any natural number to its successor. Actually, the notation `3` is only a notation for `S (S (S O))`.

New theorems can be loaded from already proven packages using the `Require` command. For example, for proofs in arithmetics, it is useful to load the following packages:

```
Require Import Arith.
Require Import ArithRing.
Require Import Omega.

Search le.
between_le: forall (P : nat -> Prop) (k l : nat),
   between P k l -> k <= l
exists_le_S:
  forall (Q : nat -> Prop) (k l : nat),
      exists_between Q k l -> S k <= l

...
plus_le_reg_l: forall n m p : nat, p + n <= p + m -> n <= m
plus_le_compat_l: forall n m p : nat, n <= m -> p + n <= p + m
plus_le_compat_r: forall n m p : nat, n <= m -> n + p <= m + p
le_plus_l: forall n m : nat, n <= n + m

...
```

There is a real notion of *false* formulas, but it is only expressed by saying that the existence of a proof for a false formula would imply the existence of a proof for the formula `False`.

A theorem that proves an implication actually is an expression whose type is a function type (in other words, an arrow type). Thus, it can be applied to a term whose type appears to the left of the arrow. From the logical point of view, the argument of this theorem should be a proof of the implication's premise. This corresponds to what is usually known as *modus ponens*.

A theorem that proves a universal quantification is also an expression whose type is a function type. It can also be applied to an argument of the right type. From the logical point of view, this corresponds to producing a new proof for the statement where the universal quantification is instantiated. Here are a few examples, where theorems are instantiated and a modus ponens inference is performed:

```
Check (le_n 0).
le_n 0 : 0 <= 0
```

```
Check (le_S 0 0).
le_S 0 1 : 0 <= 0 -> 0 <= 1

Check (le_S 0 0 (le_n 0)).
le_S 0 0 (le_n 0) : 0 <= 1
```

New theorems could be constructed this way by combining existing theorems and using the `Definition` keyword to associate these expressions to constants. But this approach is seldom used. The alternative approach is known as *goal directed proof*, with the following type of scenario:

1. the user enters a statement that he wants to prove, using the command `Theorem` or `Lemma`,
2. the Coq system displays the formula as a formula to be proved, possibly giving a context of local facts that can be used for this proof (the context is displayed above a horizontal line, the goal is displayed under the horizontal line),
3. the user enters a command to decompose the goal into simpler ones,
4. the Coq system displays a list of formulas that still need to be proved,
5. back to step 3.

Some of the commands sent at step 3 actually decrease the number of goals. When there are no more goals the proof is complete, it needs to be saved, this is performed when the user sends the command `Qed.` The commands that are especially designed to decompose goals into lists of simpler goals are called *tactics*.

Here is an example:

```
Theorem example2 : forall a b:Prop, a /\ b -> b /\ a.
1 subgoal

  ============================
   forall a b : Prop, a /\ b -> b /\ a

Proof.
 intros a b H.
1 subgoal

  a : Prop
  b : Prop
  H : a /\ b
  ============================
   b /\ a

split.
```

```
2 subgoals
...
  H : a /\ b
  ============================
   b

subgoal 2 is:
 a

elim H; intros H0 H1.
...
  H0 : a
  H1 : b
  ============================
   b

exact H1.
1 subgoal
...
  H : a /\ b
  ============================
   a

intuition.
Proof completed.

Qed.
intros a b H.
split.
 elim H; intros H0 H1.
   exact H1.
 intuition.
example2 is defined
```

This proof uses several steps to decompose the logical processing, but a quicker dialog would simply rely on the `intuition` tactic directly from the start. There is an important collection of tactics in the Coq system, each of which is adapted to a shape of goal. For instance, the tactic `elim H` was adapted because the hypothesis `H` was a proof of a conjunction of two propositions. The effect of the tactic was to add two implications in front of the goal, with two premises asserting one of the propositions in the conjunction. It is worthwhile remembering a collection of tactics for the basic logical connectives. We list these tactics in the following table, inspired from the table in [1] (p. 130).

	$\Rightarrow$	$\forall$	$\wedge$	$\vee$	$\exists$
Hypothesis	`apply`	`apply`	`elim`	`elim`	`elim`
goal	`intros`	`intros`	`split`	`left` or `right`	`exists` v
	$\neg$	$=$			
Hypothesis	`elim`	`rewrite`			
goal	`intro`	`reflexivity`			

When using the tactic `elim`, this usually creates new facts that are placed in the result goal as premises of newly created implications. These premises must then be introduced in the context using the `intros` tactic. A quicker tactic does the two operations at once, this tactic is called `destruct`.

Some automatic tactics are also provided for a variety of purposes, `intuition` is often useful to prove facts that are tautologies in first-order intuitionistic logic (try it whenever the proof only involves manipulations of forall quantification, conjunction, disjunction, and negation); `auto` is an extensible tactic that tries to apply a collection of theorems that were provided beforehand by the user, `eauto` is like `auto`, it is more powerful but also more time-consuming, `ring` and `ring_nat` mostly do proofs of equality for expressions containing addition and multiplication (and `S` for `ring_nat`), `omega` proves formulas in Presburger arithmetic.

One of the difficult points for newcomers is that the Coq system also provides a type `bool` with two elements called `true` and `false`. Actually this type has nothing to do with truth or provability, it is just a two-element type that may be used to model the boolean types that one usually finds in programming languages and the value attached to its two elements is purely conventional. In a sense, it is completely correct (but philosophically debatable) to define a function named `is_zero` that takes an argument of type `nat` and returns `false` if, and only if, its argument is 0.

4 Inductive types

Inductive types could also be called algebraic types or initial algebras. They are defined by providing the type name, its type, and a collection of constructors. Inductive types can be parameterized and dependent. We will mostly use parameterization to represent polymorphism and dependence to represent logical properties.

4.1 Defining inductive types

Here is an example of an inductive type definition:

```
Inductive bin : Set :=
  L : bin
| N : bin -> bin -> bin.
```

This defines a new type `bin`, whose type is `Set`, and provides two ways to construct elements of this type: `L` (a constant) and `N` (a function taking two arguments). The Coq system

automatically associates a theorem to this inductive type. This theorem makes it possible to reason by induction on elements of this type:

```
Check bin_ind.
bin_ind
     : forall P : bin -> Prop,
       P L ->
       (forall b : bin,
         P b -> forall b0 : bin, P b0 -> P (N b b0)) ->
       forall b : bin, P b
```

The induction theorem associated to an inductive type is always named *name*_`ind`.

4.2 Pattern matching

Elements of inductive types can be processed using functions that perform some pattern-matching. For instance, we can write a function that returns the boolean value `false` when its argument is `N L L` and returns `true` otherwise.

```
Definition example3 (t : bin): bool :=
  match t with N L L => false | _ => true end.
```

4.3 Recursive function definition

There are an infinity of different trees in the type `bin`. To write interesting functions with arguments from this type, we need more than just pattern matching. The Coq system provides recursive programming. The shape of recursive function definitions is as follows:

```
Fixpoint flatten_aux (t1 t2:bin) {struct t1} : bin :=
  match t1 with
     L => N L t2
  | N t'1 t'2 => flatten_aux t'1 (flatten_aux t'2 t2)
  end.

Fixpoint flatten (t:bin) : bin :=
  match t with
    L => L
  | N t1 t2 => flatten_aux t1 (flatten t2)
  end.

Fixpoint size (t:bin) : nat :=
  match t with
    L => 1
  | N t1 t2 => 1 + size t1 + size t2
  end.
```

There are constraints in the definition of recursive definitions. First, if the function has more than one argument, we must declare one of these arguments as the *principal* or *structural* argument (we did this declaration for the function `flatten_aux`). If there is only one argument, then this argument is the principal argument by default. Second, every recursive call must be performed so that the principal argument of the recursive call is a subterm, obtained by pattern-matching, of the initial principal argument. This condition is satisfied in all the examples above.

4.4 Proof by cases

Now, that we have defined functions on our inductive type, we can prove properties of these functions. Here is a first example, where we perform a few case analyses on the elements of the type `bin`.

```
Theorem example3_size :
   forall t, example3 t = false -> size t = 3.
Proof.
intros t; destruct t.
2 subgoals

  ============================
   example3 L = false -> size L = 3

subgoal 2 is:
 example3 (N t1 t2) = false -> size (N t1 t2) = 3
```

The tactic `destruct t` actually observes the various possible cases for `t` according to the inductive type definition. The term `t` can only be either obtained by `L`, or obtained by `N` applied to two other trees `t1` and `t2`. This is the reason why there are two subgoals.

We know the value that `example3` and `size` should take for the tree `L`. We can direct the Coq system to compute it:

```
simpl.
2 subgoals

  ============================
   true = false -> 1 = 3
```

After computation, we dicover that assuming that the tree is `L` and that the value of `example3` for this tree is `false` leads to an inconsistency. We can use the following tactics to exploit this kind of inconsistency:

```
intros H.
  H : true = false
```

```
  ============================
   1 = 3
discriminate H.
1 subgoal
...
  ============================
   example3 (N t1 t2) = false -> size (N t1 t2) = 3
```

The answer shows that the first goal was solved. The tactic `discriminate H` can be used whenever the hypothesis `H` is an assumption that asserts that two different constructors of an inductive type return equal values. Such an assumption is inconsistent and the tactic exploits directly this inconsistency to express that the case described in this goal can never happen. This tactic expresses a basic property of inductive types in the sort `Set` or `Type`: constructors have distinct ranges. Another important property of constructors of inductive types in the sort `Set` or `Type` is that they are injective. The tactic to exploit this fact called `injection` (for more details about `discriminate` and `injection` please refer to [1] or the Coq reference manual [2]).

For the second goal we still must do a case analysis on the values of `t1` and `t2`, we do not detail the proof but it can be completed with the following sequence of tactics.

```
destruct t1.
destruct t2.
3 subgoals

  ============================
   example3 (N L L) = false -> size (N L L) = 3

subgoal 2 is:
 example3 (N L (N t2_1 t2_2)) = false ->
 size (N L (N t2_1 t2_2)) = 3
subgoal 3 is:
 example3 (N (N t1_1 t1_2) t2) = false ->
  size (N (N t1_1 t1_2) t2) = 3
```

For the first goal, we know that both functions will compute as we stated by the equality's right hand side. We can solve this goal easily, for instance with `auto`. The last two goals are solved in the same manner as the very first one, because `example3` cannot possibly have the value `false` for the arguments that are given in these goals.

```
auto.
intros H; discriminate H.
intros H; discriminate H.
Qed.
```

To perform this proof, we have simply observed 5 cases. For general recursive functions, just observing a finite number of cases is not sufficient. We need to perform proofs by induction.

4.5 Proof by induction

The most general kind of proof that one can perform on inductive types is proof by induction.

When we prove a property of the elements of an inductive type using a proof by induction, we actually consider a case for each constructor, as we did for proofs by cases. However there is a twist: when we consider a constructor that has arguments of the inductive type, we can assume that the property we want to establish holds for each of these arguments.

When we do goal directed proof, the induction principle is invoked by the `elim` tactic. To illustrate this tactic, we will prove a simple fact about the `flatten_aux` and `size` functions.

```
Theorem forall_aux_size :
 forall t1 t2, size (flatten_aux t1 t2) = size t1 + size t2 + 1.
Proof.
 intros t1; elim t1.
  ============================
   forall t2 : bin, size (flatten_aux L t2) = size L + size t2 + 1

subgoal 2 is:
 forall b : bin,
 (forall t2 : bin, size (flatten_aux b t2) =
                   size b + size t2 + 1) ->
 forall b0 : bin,
 (forall t2 : bin, size (flatten_aux b0 t2) =
                   size b0 + size t2 + 1) ->
 forall t2 : bin,
 size (flatten_aux (N b b0) t2) =
 size (N b b0) + size t2 + 1
```

There are two subgoals, the first goal requires that we prove the property when the first argument of `flatten_aux` is `L`, the second one requires that we prove the property when the argument is `N b b0`, under the assumption that it is already true for `b` and `b0`. The proof progresses easily, using the definitions of the two functions, which are expanded when the Coq system executes the `simpl` tactic. We then obtain expressions that can be solved using the `ring_nat` tactic.

```
intros t2.
simpl.
...
  ============================
   S (S (size t2)) = S (size t2 + 1)
```

```
ring_nat.
intros b IHb b0 IHb0 t2.
...
  IHb : forall t2 : bin, size (flatten_aux b t2) =
                                  size b + size t2 + 1
  b0 : bin
  IHb0 : forall t2 : bin, size (flatten_aux b0 t2) =
                          size b0 + size t2 + 1
  t2 : bin
  ============================
   size (flatten_aux (N b b0) t2) = size (N b b0) + size t2 + 1

simpl.
...
  ============================
   size (flatten_aux b (flatten_aux b0 t2)) =
   S (size b + size b0 + size t2 + 1)

rewrite IHb.
...
  ============================
   size b + size (flatten_aux b0 t2) + 1 =
   S (size b + size b0 + size t2 + 1)

rewrite IHb0.
...
  ============================
   size b + (size b0 + size t2 + 1) + 1 =
   S (size b + size b0 + size t2 + 1)

ring_nat.
Proof completed.
Qed.
```

4.6 Numbers in the Coq system

In the Coq system, most usual datatypes are represented as inductive types and packages provide a variety of properties, functions, and theorems around these datatypes. The package named `Arith` contains a host of theorems about natural numbers (numbers from 0 to infinity), which are described as an inductive type with `O` (representing 0) and `S` as constructors. It also provides addition, multiplication, subtraction (with the special behavior that `x - y` is 0 when `x` is smaller than `y`. The package `ArithRing` contains the tactic `ring_nat` and the associated theorems.

The package named `ZArith` provides two inductive datatypes to represent integers. The

first inductive type, named `positive`, follows a binary representation to model the positive integers (from 1 to infinity) and the type `Z` is described as a type with three constructors, one for positive numbers, one for negative numbers, and one for 0. The package also provides orders and basic operations: addition, subtraction, multiplication, division, square root. The package `ZArithRing` provides the configuration for the `ring` tactic to work on polynomial equalities with numbers of type `Z`. The tactic `omega` works equally well to solve problems in Presburger arithmetic for both natural numbers of type `nat` and integers of type `Z`.

There are a few packages that provide descriptions of rational numbers, but the support for these packages is not as standard as for the other one. For instance, there is no easy notation to enter a simple fraction. In a paper from 2001, I showed that the rational numbers could be given a very simple inductive structure, but encoding the various basic operations on this structured was a little complex.

The support for computation on real numbers in the Coq system is also quite good, but real numbers are not (and cannot) be represented using an inductive type. The package to load is `Reals` and it provides descriptions for quite a few functions, up to trigonomic functions for instance.

4.7 Data-structures

The two-element boolean type is an inductive type in the Coq system, `true` and `false` are its constructors. The induction principle naturally expresses that this type only has two elements, because it suffices that a property is satisfied by `true` and `false` to ensure that it is satisfied by all elements of `bool`. On the other hand, it is easy to prove that `true` and `false` are distinct.

Most ways to structure data together are also provided using inductive data structures. The pairing construct actually is an inductive type, and `elim` can be used to reason about a pair in the same manner as it can be used to reason on a natural number.

A commonly used datatype is the type of lists. This type is polymorphic, in the sense that the same inductive type can be used for lists of natural numbers, lists of boolean values, or lists of other lists. This type is not provided by default in the Coq system, it is necessary to load the package `List` using the `Require` command to have access to it. The usual `cons` function is given in Coq as a three argument function, where the first argument is the type of elements: it is the type of the second argument and the third argument should be a list of elements of this type, too. The empty list is represented by a function `nil`. This function also takes an argument, which should be a type. However, the Coq system also provides a notion of implicit arguments, so that the type arguments are almost never written and the Coq system infers them from the context or the other arguments. For instance, here is how we construct a list of natural numbers.

```
Require Import List.

Check (cons 3 (cons 2 (cons 1 nil))).
3 :: 2 :: 1 :: nil : list nat
```

This example also shows that the notation `::` is used to represent the cons function in an infix fashion. This tradition will be very comfortable to programmers accustomed to languages

in the ML family, but Haskell addicts should beware that the conventions, between type information and cons, are inverse to the conventions in Haskell.

The `List` package also provides a list concatenation function named `app`, with `++` as infix notation, and a few theorems about this function.

5 Inductive properties

Inductive types can be dependent and when they are, they can be used to express logical properties. When defining inductive types like `nat`, `Z` or `bool` we declare that this constant is a type. When defining a dependent type, we actually introduce a new constant which is declared to be a function from some input type to a type of types. Here is an example:

```
Inductive even : nat -> Prop :=
  even0 : even 0
| evenS : forall x:nat, even x -> even (S (S x)).
```

Thus, `even` itself is not a type, it is `even x`, whenever `x` is an integer that is a type. In other words, we actually defined a family of types. In this family, not all members contain elements. For instance, we know that the type `even 0` contains an element, this element is `even0`, and we know that `even 2` contains an element: `evenS 0 even0`. What about `even 1`? If `even 1` contains an element, this element cannot be obtained using `even0` because $0 \neq 1$, it cannot be obtained using `evenS` because $1 \neq 2 + x$ for every x such that $0 \leq x$. Pushing our study of `even` further we could see that this type, seen as a property, is provable if and only if its argument is even, in the common mathematical sense.

Like other inductive types, inductive properties are equipped with an induction principle, which we can use to perform proofs. The inductive principle for `even` has the following shape.

```
Check even_ind.
even_ind : forall P : nat -> Prop,
       P 0 ->
       (forall x : nat, even x -> P x -> P (S (S x))) ->
       forall n : nat, even n -> P n
```

This principle intuitively expresses that `even` is the smallest property satisfying the two constructors: it implies every other property that also satisfies them. A proof using this induction principle will work on a goal where we know that `even y` holds for some `y` and actually decomposes into a proof of two cases, one corresponding to the case where `even y` was obtained using `even0` and one corresponding to the case where `even y` was obtained using `evenS`, applied to some `x` such that `y=S (S x)` and some proof of `even y`. In the second case, we again have the opportunity to use an induction hypothesis about this `y`.

When a variable `x` satisfies an inductive property, it is often more efficient to prove properties about this variable using an induction on the inductive property than an induction on the variable itself. The following proof is an example:

```
Theorem even_mult : forall x, even x -> exists y, x = 2*y.
Proof.
intros x H; elim H.
2 subgoals

  x : nat
  H : even x
  ============================
   exists y : nat, 0 = 2 * y

subgoal 2 is:
 forall x0 : nat,
 even x0 -> (exists y : nat, x0 = 2 * y) ->
 exists y : nat, S (S x0) = 2 * y

exists 0; ring_nat.
intros x0 Hevenx0 IHx.
...
  IHx : exists y : nat, x0 = 2 * y
  ============================
   exists y : nat, S (S x0) = 2 * y
```

In the last goal, `IHx` is the induction hypothesis. It says that if `x0` is the predecessor's predecessor of `x` then we already know that there exists a value `y` that is its half. We can use this value to provide the half of `S (S x0)`. Here are the tactics that complete the proof.

```
destruct IHx as [y Heq]; rewrite Heq.
exists (S y); ring_nat.
Qed.
```

In this example, we used a variant of the `destruct` tactic that makes it possible to choose the name of the elements that `destruct` creates and introduces in the context.

If we wanted to prove the same property using a direct induction on the natural number that is even, we would have a problem because the predecessor of an even number, for which direct induction provides an induction hypothesis, is not even. The proof is not impossible but slightly more complex:

```
Theorem even_mult' : forall x, even x -> exists y, x = 2* y.
Proof.
intros x.
assert (lemma: (even x -> exists y, x=2*y)/\
        (even (S x) -> exists y, S x=2*y)).
elim x.
split.
exists 0; ring_nat.
```

```
intros Heven1; inversion Heven1.
intros x0 IHx0; destruct IHx0 as [IHx0 IHSx0].
split.
exact IHSx0.
intros HevenSSx0.
assert (Hevenx0 : even x0).
inversion HevenSSx0; assumption.
destruct (IHx0 Hevenx0) as [y Heq].
rewrite Heq; exists (S y); ring_nat.
intuition.
Qed.
```

This script mostly uses tactics that we have already introduced, except the `inversion` tactic. Given an assumption `H` that relies on a dependent inductive type, most frequently an inductive proposition, the tactic `inversion` analyses all the constructors of the inductive, discards the ones that could not have been applied, and when some constructors could have applied it creates a new goal where the premises of this constructor are added in the context. For instance, this tactic is perfectly suited to prove that 1 is not even:

```
Theorem not_even_1 : ~even 1.
Proof.
intros even1.
...
  even1 : even 1
  ============================
   False

inversion even1.
Qed.
```

This example also shows that the negation of a fact actually is represented by a function that says "this fact implies `False`".

Inductive properties can be used to express very complex notions. For instance, the semantics of a programming language can be defined as an inductive definition, using dozens of constructors, each one describing a kind of computation elementary step. Proofs by induction with respect to this inductive definition correspond to what Wynskel calls *rule induction* in his introductory book on programming language semantics [3].

6 Exercices

Most of the exercices proposed here are taken from [1].

Exercise 5.6 Prove the following theorems:

```
forall A B C:Prop, A/\(B/\C)→(A/\B)/\C
```

```
forall A B C D: Prop,(A→B)/\(C→D)/\A/\C → B/\D
forall A: Prop, ~(A/\~A)
forall A B C: Prop, A\/(B\/C)→(A\/B)\/C
forall A: Prop, ~~(A\/~A)
forall A B: Prop, (A\/B)/\~A → B
```

Two benefits can be taken from this exercise. In a first step you should try using only the basic tactics given in the table page 7. In a second step, you can verify which of these statements are directly solved by the tactic `intuition`.

Universal quantification Prove

```
forall A:Set,forall P Q:A→Prop,
    (forall x, P x)\/(forall y, Q y)→forall x, P x\/Q y.
~(forall A:Set,forall P Q:A→Prop,
   (forall x, P x\/Q x)→(forall x, P x)\/(forall y, Q y).
```

Hint: for the second exercise, think about a counter-example with the type `bool` and its two elements `true` and `false`, which can be proved different, for instance.

Exercise 6.32 The sum of the first n natural numbers is defined with the following function:

```
Fixpoint sum_n (n:nat) : nat :=
  match n with  0 => 0 | S p => S p + sum_n p end.
```

Prove the following statement:

```
forall n:nat, 2 * sum_n n = n*n + n
```

Square root of 2 If $p^2 = 2q^2$, then $(2q-p)^2 = 2(p-q)^2$, now if p is the least positive integer such that there exists a positive integer q such that $p/q = \sqrt{2}$, then $p > 2q - p > 0$, and $(2q-p)/(p-q) = \sqrt{2}$. This is a contradiction and a proof that $\sqrt{2}$ is not rational. Use Coq to verify a formal proof along these lines.

7 Solutions

The solutions to the numbered exercises are available from the internet (on the site associated to the reference [1]). The short proof that $\sqrt{2}$ is not rational is also available on the internet from the author's personal web-page.

Here are the solutions to the exercises on universal quantification.

```
Theorem ex1 :
  forall A:Set, forall P Q:A->Prop,
  (forall x, P x) \/ (forall y, Q y) -> forall x, P x \/ Q x.
Proof.
```

```
 intros A P Q H.
 elim H.
 intros H1; left; apply H1.
 intros H2; right; apply H2.
Qed.

Theorem ex2 :
  ~(forall A:Set, forall P Q:A->Prop,
    (forall x, P x \/ Q x) -> (forall x, P x) \/ (forall y, Q y)).
Proof.
  intros H; elim (H bool (fun x:bool => x = true)
                    (fun x:bool => x = false)).
  intros H1; assert (H2:false = true).
    apply H1.
  discriminate H2.
  intros H1; assert (H2:true = false).
    apply H1.
  discriminate H2.
  intros x; case x.
  left; reflexivity.
  right; reflexivity.
Qed.
```

References

[1] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art:the Calculus of Inductive Constructions*. Springer-Verlag, 2004.

[2] The Coq development team. *The Coq proof Assistant Reference Manual*, Ecole Polytechnique, INRIA, Universit de Paris-Sud, 2004. `http://coq.inria.fr/doc/main.html`

[3] G. Winskel. *The Formal Semantics of Programming Languages, an introduction*. Foundations of Computing. The MIT Press, 1993.

Tobias Nipkow:

Isabelle/HOL

A Compact Introduction to Isabelle/HOL

Tobias Nipkow

TU München

Overview

1. Introduction
2. Datatypes
3. Logic
4. Sets

Overview of Isabelle/HOL

System Architecture

ProofGeneral	(X)Emacs based interface
Isabelle/HOL	Isabelle instance for HOL
Isabelle	generic theorem prover
Standard ML	implementation language

HOL

HOL = Higher-Order Logic

HOL = Functional programming + Logic

HOL has

- datatypes
- recursive functions
- logical operators ($\wedge$, $\longrightarrow$, $\forall$, $\exists$, ...)

HOL is a programming language!

Higher-order = functions are values, too!

Formulae

Syntax (in decreasing priority):

$$\begin{array}{rcllll} form & ::= & (form) & | & term = term & | \ \neg form \\ & | & form \wedge form & | & form \vee form & | \ form \longrightarrow form \\ & | & \forall x.\ form & | & \exists x.\ form & \end{array}$$

Scope of quantifiers: as far to the right as possible

Examples

- $\neg A \wedge B \vee C \equiv ((\neg A) \wedge B) \vee C$
- $A = B \wedge C \equiv (A = B) \wedge C$
- $\forall x.\ P\ x \wedge Q\ x \equiv \forall x.\ (P\ x \wedge Q\ x)$
- $\forall x.\ \exists y.\ P\ x\ y \wedge Q\ x \equiv \forall x.\ (\exists y.\ (P\ x\ y \wedge Q\ x))$

Types and Terms

Types

Syntax:

τ	::=	(τ)	
	|	*bool* | *nat* | ...	base types
	|	*'a* | *'b* | ...	type variables
	|	$\tau \Rightarrow \tau$	total functions
	|	$\tau \times \tau$	pairs (ascii: *)
	|	τ *list*	lists
	|	...	user-defined types

Parentheses: *T1* $\Rightarrow$ *T2* $\Rightarrow$ *T3* $\equiv$ *T1* $\Rightarrow$ *(T2* $\Rightarrow$ *T3)*

Terms: Basic syntax

Syntax:

$term$	::=	$(term)$	
	|	a	constant or variable (identifier)
	|	$term\ term$	function application
	|	$\lambda x.\ term$	function "abstraction"
	|	...	lots of syntactic sugar

Examples: *f (g x) y* *h (*λ*x. f (g x))*

Parantheses: *f* a_1 a_2 a_3 $\equiv$ *((f* a_1*)* a_2*)* a_3

Terms and Types

Terms must be well-typed

(the argument of every function call must be of the right type)

Notation: $t :: \tau$ means t is a well-typed term of type τ.

Type inference

Isabelle automatically computes ("*infers*") the type of each variable in a term.

In the presence of *overloaded* functions (functions with multiple types) not always possible.

User can help with type annotations inside the term.

Example: *f (x::nat)*

Currying

Thou shalt curry your functions

- Curried: *f* :: $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau$
- Tupled: *f'* :: $\tau_1 \times \tau_2 \Rightarrow \tau$

Advantage: *partial application* *f* a_1 with $a_1 :: \tau_1$

Terms: Syntactic sugar

Some predefined syntactic sugar:

- *Infix:* +, -, *, #, @, . . .
- *Mixfix: if _ then _ else _*, *case _ of*, . . .

Prefix binds more strongly than infix:

! $f\ x + y \equiv (f\ x) + y \not\equiv f\ (x + y)$!

Base types: bool, nat, list

Type bool

Formulae = terms of type *bool*

True :: bool
False :: bool
$\wedge, \vee, \ldots$ *:: bool* $\Rightarrow$ *bool* $\Rightarrow$ *bool*
⋮

if-and-only-if: =

Type nat

0 :: nat
Suc :: nat ⇒ nat
*+, *, ... :: nat ⇒ nat ⇒ nat*
⋮

! Numbers and arithmetic operations are overloaded:
0,1,2,... :: 'a, + :: 'a ⇒ 'a ⇒ 'a

You need type annotations: *1 :: nat, x + (y::nat)*

. . . unless the context is unambiguous: *Suc z*

Type list

- *[]*: empty list
- *x # xs*: list with first element *x* ("*head*") and rest *xs* ("*tail*")
- Syntactic sugar: $[x_1,\ldots,x_n]$

Large library:
hd, *tl*, *map*, *size*, *filter*, *set*, *nth*, *take*, *drop*, *distinct*, . . .

Don't reinvent, reuse!
⇝ `HOL/List.thy`

Isabelle Theories

Theory = Module

Syntax:

`theory` $MyTh$ `=` $ImpTh_1$ `+` `...` `+` $ImpTh_n$`:`
(declarations, definitions, theorems, proofs, ...)*
`end`

- $MyTh$: name of theory. Must live in file $MyTh$`.thy`
- $ImpTh_i$: name of *imported* theories. Import transitive.

Unless you need something special:

`theory` $MyTh$ `= Main:`

Proof General

An Isabelle Interface

by David Aspinall

Proof General

Customized version of (x)emacs:

- all of emacs (info: `C-h i`)
- Isabelle aware (when editing `.thy` files)
- mathematical symbols (“x-symbols”)

Interaction:

- via mouse
- or keyboard (key bindings see `C-h m`)

X-Symbols

Input of funny symbols in Proof General

- via menu (“X-Symbol”)
- via ascii encoding (similar to LaTeX): `\<and>`, `\<or>`, ...
- via abbreviation: `/\`, `\/`, `-->`, ...

x-symbol	∀	∃	λ	¬	∧	∨	⟶	⇒
ascii (1)	`\<forall>`	`\<exists>`	`\<lambda>`	`\<not>`	`/\`	`\/`	`-->`	`=>`
ascii (2)	`ALL`	`EX`	`%`	`~`	`&`	`|`		

(1) is converted to x-symbol, (2) stays ascii.

Demo: terms and types

An introduction to recursion and induction

A recursive datatype: toy lists

datatype *'a list = Nil | Cons 'a "'a list"*

***Nil*:** empty list

***Cons x xs*:** head *x :: 'a*, tail *xs :: 'a list*

A toy list: *Cons False (Cons True Nil)*

Predefined lists: *[False, True]*

Concrete syntax

In `.thy` files:
Types and formulae need to be inclosed in *"..."*

Except for single identifiers, e.g. *'a*

"..." normally not shown on slides

Structural induction on lists

P xs holds for all lists *xs* if

- *P Nil*
- and for arbitrary *x* and *xs*, *P xs* implies *P (Cons x xs)*

Demo: append and reverse

Proofs

General schema:

lemma *name*`: "..."`
apply `(...)`
apply `(...)`
⋮
done

If the lemma is suitable as a simplification rule:

lemma *name*`[simp]: "..."`

Proof methods

- Structural induction
  - Format: *(induct x)*
    x must be a free variable in the first subgoal.
    The type of *x* must be a datatype.
  - Effect: generates 1 new subgoal per constructor
- Simplification and a bit of logic
  - Format: *auto*
  - Effect: tries to solve as many subgoals as possible using simplification and basic logical reasoning.

The proof state

$1. \bigwedge x_1 \ldots x_p. \llbracket A_1; \ldots ; A_n \rrbracket \Longrightarrow B$

$x_1 \ldots x_p$	Local constants
$A_1 \ldots A_n$	Local assumptions
B	Actual (sub)goal

Notation

$\llbracket A_1; \ldots ; A_n \rrbracket \Longrightarrow B$

abbreviates

$A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow B$

$;$ $\approx$ "and"

Type and function definition in Isabelle/HOL

Datatype definition in Isabelle/HOL

The example

datatype *'a list = Nil | Cons 'a "'a list"*

Properties:

- Types: *Nil :: 'a list*
  Cons :: 'a ⇒ 'a list ⇒ 'a list
- Distinctness: *Nil ≠ Cons x xs*
- Injectivity: *(Cons x xs = Cons y ys) = (x = y ∧ xs = ys)*

The general case

datatype $(\alpha_1, \ldots, \alpha_n)\tau$ = $C_1\ \tau_{1,1} \ldots \tau_{1,n_1}$
| $\ldots$
| $C_k\ \tau_{k,1} \ldots \tau_{k,n_k}$

- *Types:* $C_i :: \tau_{i,1} \Rightarrow \cdots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)\tau$
- *Distinctness:* $C_i\ \ldots \neq C_j\ \ldots$ if $i \neq j$
- *Injectivity:*
  $(C_i\ x_1 \ldots x_{n_i} = C_i\ y_1 \ldots y_{n_i}) = (x_1 = y_1 \wedge \ldots \wedge x_{n_i} = y_{n_i})$

Distinctness and Injectivity are applied automatically
Induction must be applied explicitly

case

Every datatype introduces a *case* construct, e.g.

(case xs of [] ⇒ . . . | y#ys ⇒ ... y ... ys ...)

In general: one case per constructor

Same order of cases as in datatype

No nested patterns (e.g. *x#y#zs*)

But nested cases

Needs *( )* in context

Case distinctions

apply(*case_tac* t)

creates k subgoals

$$t = C_i\ x_1 \ldots x_p \Longrightarrow \ldots$$

one for each constructor C_i.

Function definition in Isabelle/HOL

Why nontermination can be harmful

How about $f\ x = f\ x + 1$?

Subtract $f\ x$ on both sides.

$$\Longrightarrow 0 = 1$$

! All functions in HOL must be total !

Function definition schemas in Isabelle/HOL

- Non-recursive with **defs/constdefs**
  No problem
- Primitive-recursive with **primrec**
  Terminating by construction
- Well-founded recursion with **recdef**
  User must (help to) prove termination
  ($\rightsquigarrow$ later)

primrec

The example

primrec
"app Nil ys = ys"

"app (Cons x xs) ys = Cons x (app xs ys)"

The general case

If τ is a datatype (with constructors $C_1, \ldots, C_k$) then
$f :: \cdots \Rightarrow \tau \Rightarrow \cdots \Rightarrow \tau'$ can be defined by *primitive recursion*:

$$f\ x_1 \ldots (C_1\ y_{1,1} \ldots y_{1,n_1}) \ldots x_p \quad = \quad r_1$$
$$\vdots$$
$$f\ x_1 \ldots (C_k\ y_{k,1} \ldots y_{k,n_k}) \ldots x_p \quad = \quad r_k$$

The recursive calls in r_i must be *structurally smaller*,
i.e. of the form $f\ a_1 \ldots y_{i,j} \ldots a_p$

nat is a datatype

datatype *nat = 0 | Suc nat*

Functions on *nat* definable by primrec!

primrec
f 0 = ...
f(Suc n) = ... f n ...

Demo: trees

Proof by Simplification

Term rewriting foundations

Term rewriting means . . .

Using equations $l = r$ from left to right

As long as possible

Terminology: equation $\leadsto$ *rewrite rule*

An example

Equations:

$$\begin{array}{rcll} 0 + n & = & n & (1) \\ (Suc\ m) + n & = & Suc\ (m + n) & (2) \\ (Suc\ m \leq Suc\ n) & = & (m \leq n) & (3) \\ (0 \leq m) & = & True & (4) \end{array}$$

Rewriting:

$$\begin{array}{rcll} 0 + Suc\ 0 & \leq & Suc\ 0 + x & \stackrel{(1)}{=} \\ Suc\ 0 & \leq & Suc\ 0 + x & \stackrel{(2)}{=} \\ Suc\ 0 & \leq & Suc\ (0 + x) & \stackrel{(3)}{=} \\ 0 & \leq & 0 + x & \stackrel{(4)}{=} \\ & & True & \end{array}$$

Interlude: Variables in Isabelle

Schematic variables

Three kinds of variables:

- bound: $\forall x.\ x = x$
- free: $x = x$
- schematic: $?x = ?x$ (“unknown”)

Can be mixed: $\forall b.\ f\ ?a\ y = b$

- Logically: free = schematic
- Operationally:
  - free variables are fixed
  - schematic variables are instantiated by substitutions (e.g. during rewriting)

From x to ?x

State lemmas with free variables:

lemma *app_Nil2[simp]: "xs @ [] = xs"*

⋮

done

After the proof: Isabelle changes *xs* to *?xs* (internally):

?xs @ [] = ?xs

Now usable with arbitrary values for *?xs*

Term rewriting in Isabelle

Basic simplification

Goal: *1.* $[\![P_1; \ldots ; P_m]\!] \Longrightarrow C$

apply(*simp add:* $eq_1 \ldots eq_n$)

Simplify $P_1 \ldots P_m$ and C using

- lemmas with attribute *simp*
- rules from **primrec** and **datatype**
- additional lemmas $eq_1 \ldots eq_n$
- assumptions $P_1 \ldots P_m$

auto versus simp

- *auto* acts on all subgoals
- *simp* acts only on subgoal 1
- *auto* applies *simp* and more

Termination

Simplification may not terminate.
Isabelle uses *simp*-rules (almost) blindly from left to right.

Conditional *simp*-rules are only applied
if conditions are provable.

Demo: simp

Induction heuristics

Basic heuristics

Theorems about recursive functions are proved by induction

Induction on argument number i of f
if f is defined by recursion on argument number i

A tail recursive reverse

consts *itrev :: 'a list ⇒ 'a list ⇒ 'a list*
primrec
itrev [] ys = ys
itrev (x#xs) ys = itrev xs (x#ys)

lemma *itrev xs [] = rev xs*

Why in this direction?

Because the lhs is “more complex” than the rhs.

Demo: first proof attempt

Generalisation (1)

Replace constants by variables

lemma *itrev xs ys = rev xs @ ys*

Demo: second proof attempt

Generalisation (2)

Quantify free variables by ∀
(except the induction variable)

lemma ∀*ys. itrev xs ys = rev xs @ ys*

HOL: Propositional Logic

Overview

- Natural deduction
- Rule application in Isabelle/HOL

Rule notation

$$\frac{A_1 \ldots A_n}{A}$$ instead of $$[\![A_1 \ldots A_n]\!] \Longrightarrow A$$

Natural Deduction

Natural deduction

Two kinds of rules for each logical operator $\oplus$:

Introduction: how can I prove $A \oplus B$?

Elimination: what can I prove from $A \oplus B$?

Natural deduction for propositional logic

$$\frac{A \quad B}{A \wedge B}\ \texttt{conjI} \qquad \frac{A \wedge B \quad [\![A;B]\!] \Longrightarrow C}{C}\ \texttt{conjE}$$

$$\frac{A}{A \vee B}\ \frac{B}{A \vee B}\ \texttt{disjI1/2} \qquad \frac{A \vee B \quad A \Longrightarrow C \quad B \Longrightarrow C}{C}\ \texttt{disjE}$$

$$\frac{A \Longrightarrow B}{A \longrightarrow B}\ \texttt{impI} \qquad \frac{A \longrightarrow B \quad A \quad B \Longrightarrow C}{C}\ \texttt{impE}$$

$$\frac{A \Longrightarrow B \quad B \Longrightarrow A}{A = B}\ \texttt{iffI} \qquad \frac{A=B}{A \Longrightarrow B}\ \texttt{iffD1} \quad \frac{A=B}{B \Longrightarrow A}\ \texttt{iffD2}$$

$$\frac{A \Longrightarrow \mathit{False}}{\neg A}\ \texttt{notI} \qquad \frac{\neg A \quad A}{C}\ \texttt{notE}$$

Operational reading

$$\frac{A_1 \ldots A_n}{A}$$

Introduction rule:

To prove A it suffices to prove $A_1 \ldots A_n$.

Elimination rule

If I know A_1 and want to prove A
it suffices to prove $A_2 \ldots A_n$.

Classical contradiction rules

$$\frac{\neg A \Longrightarrow \mathit{False}}{A}\ \texttt{ccontr} \qquad \frac{\neg A \Longrightarrow A}{A}\ \texttt{classical}$$

Proof by assumption

$$\frac{A_1 \quad \ldots \quad A_n}{A_i}\ \texttt{assumption}$$

Rule application: the rough idea

Applying rule $[\![A_1; \ldots ; A_n]\!] \Longrightarrow A$ to subgoal C:

- Unify A and C
- Replace C with n new subgoals $A_1 \ldots A_n$

Working backwards, like in Prolog!

Example: rule: $[\![?P; ?Q]\!] \Longrightarrow ?P \wedge ?Q$
subgoal: $1.\ A \wedge B$

Result: $1.\ A$
$2.\ B$

Rule application: the details

Rule:	$[\![A_1; \ldots ; A_n]\!] \Longrightarrow A$
Subgoal:	$1.\ [\![B_1; \ldots ; B_m]\!] \Longrightarrow C$
Substitution:	$\sigma(A) \equiv \sigma(C)$
New subgoals:	$1.\ \sigma([\![B_1; \ldots ; B_m]\!] \Longrightarrow A_1)$
	$\vdots$
	$n.\ \sigma([\![B_1; \ldots ; B_m]\!] \Longrightarrow A_n)$

Command:

apply*(rule <rulename>)*

Proof by assumption

apply *assumption*

proves

$1.\ [\![B_1; \ldots ; B_m]\!] \Longrightarrow C$

by unifying C with one of the B_i (backtracking!)

Demo: application of introduction rule

Applying elimination rules

apply(*erule <elim-rule>*)

Like *rule* but also

- unifies first premise of rule with an assumption
- eliminates that assumption

Example:

Rule:	$[\![?P \wedge ?Q;\ [\![?P;\ ?Q]\!] \Longrightarrow ?R]\!] \Longrightarrow ?R$
Subgoal:	$1.\ [\![\ X;\ A \wedge B;\ Y\ ]\!] \Longrightarrow Z$
Unification:	$?P \wedge ?Q \equiv A \wedge B$ and $?R \equiv Z$
New subgoal:	$1.\ [\![\ X;\ Y\ ]\!] \Longrightarrow [\![\ A;\ B\ ]\!] \Longrightarrow Z$
same as:	$1.\ [\![\ X;\ Y;\ A;\ B\ ]\!] \Longrightarrow Z$

How to prove it by natural deduction

- Intro rules decompose formulae to the right of $\Longrightarrow$.

  apply(*rule <intro-rule>*)

- Elim rules decompose formulae on the left of $\Longrightarrow$.

  apply(*erule <elim-rule>*)

Demo: examples

$\Longrightarrow$ vs $\longrightarrow$

- Write theorems as $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$
  not as $A_1 \wedge \ldots \wedge A_n \longrightarrow A$ (to ease application)
- *Exception* (in **apply**-style): induction variable must not occur in the premises.

  Example: $[\![A; B(x)]\!] \Longrightarrow C(x) \rightsquigarrow A \Longrightarrow B(x) \longrightarrow C(x)$

  Reverse transformation (after proof):
  lemma *abc[rule_format]*: $A \Longrightarrow B(x) \longrightarrow C(x)$

Demo: further techniques

HOL: Predicate Logic

Parameters

Subgoal:

1. $\bigwedge x_1 \ldots x_n$*. Formula*

The x_i are called parameters of the subgoal.
Intuition: local constants, i.e. arbitrary but fixed values.

Rules are automatically lifted over $\bigwedge x_1 \ldots x_n$ and applied directly to *Formula*.

Scope

- Scope of parameters: whole subgoal
- Scope of $\forall$, $\exists$, ...: ends with *;* or $\Longrightarrow$

$$\bigwedge x\ y.\ [\![\ \forall y.\ P\ y \longrightarrow Q\ z\ y;\ \ Q\ x\ y\ ]\!] \Longrightarrow \exists x.\ Q\ x\ y$$

means

$$\bigwedge x\ y.\ [\![\ (\forall y_1.\ P\ y_1 \longrightarrow Q\ z\ y_1);\ \ Q\ x\ y\ ]\!] \Longrightarrow \exists x_1.\ Q\ x_1\ y$$

α-Conversion

Bound variables are renamed automatically to avoid name clashes with other variables.

Natural deduction for quantifiers

$$\frac{\bigwedge x.\ P(x)}{\forall x.\ P(x)}\ \texttt{allI} \qquad \frac{\forall x.\ P(x) \quad P(?x) \Longrightarrow R}{R}\ \texttt{allE}$$

$$\frac{P(?x)}{\exists x.\ P(x)}\ \texttt{exI} \qquad \frac{\exists x.\ P(x) \quad \bigwedge x.\ P(x) \Longrightarrow R}{R}\ \texttt{exE}$$

- `allI` and `exE` introduce new parameters ($\bigwedge x$).
- `allE` and `exI` introduce new unknowns ($?x$).

Instantiating rules

apply(*rule_tac x = "*$term$*" in* $rule$*)*

Like *rule*, but *?x* in $rule$ is instantiated by $term$ before application.

Similar: *erule_tac*

! *x* is in $rule$, not in the goal !

Two successful proofs

1. $\forall x.\ \exists y.\ x = y$
apply(*rule allI*)
1. $\bigwedge x.\ \exists y.\ x = y$

best practice

apply(*rule_tac x = "x" in exI*)
1. $\bigwedge x.\ x = x$
apply(*rule refl*)

simpler & clearer

exploration

apply(*rule exI*)
1. $\bigwedge x.\ x = ?y\ x$
apply(*rule refl*)
$?y \mapsto \lambda u.\ u$

shorter & trickier

Demo: quantifier proofs

Safe and unsafe rules

Safe `allI, exE`

Unsafe `allE, exI`

Create parameters first, unknowns later

Demo: proof methods

Sets

Overview

- Set notation
- Inductively defined sets

Set notation

Sets

Type *'a set*: sets over type *'a*

- *{}*, *{e_1,...,e_n}*, *{x. P x}*
- $e \in A$, $A \subseteq B$
- $A \cup B$, $A \cap B$, *A - B*, *- A*
- $\bigcup_{x \in A} B\ x$, $\bigcap_{x \in A} B\ x$
- *{i..j}*
- *insert :: 'a* $\Rightarrow$ *'a set* $\Rightarrow$ *'a set*
- ...

Proofs about sets

Natural deduction proofs:

- `equalityI`: $[\![A \subseteq B; B \subseteq A]\!] \Longrightarrow A = B$
- `subsetI`: $(\bigwedge x.\ x \in A \Longrightarrow x \in B) \Longrightarrow A \subseteq B$
- ... (see Tutorial)

Demo: proofs about sets

Inductively defined sets

Example: finite sets

Informally:

- The empty set is finite
- Adding an element to a finite set yields a finite set
- These are the only finite sets

In Isabelle/HOL:

consts *Fin :: 'a set set* — The set of all finite set
inductive *Fin*
intros
{} ∈ Fin
A ∈ Fin ⟹ insert a A ∈ Fin

Example: even numbers

Informally:

- 0 is even
- If n is even, so is $n + 2$
- These are the only even numbers

In Isabelle/HOL:

consts *Ev :: nat set* — The set of all even numbers
inductive *Ev*
intros
0 ∈ Ev
n ∈ Ev ⟹ n + 2 ∈ Ev

Format of inductive definitions

consts $S :: \tau\ set$
inductive S
intros
$[\![a_1 \in S; \ldots ; a_n \in S; A_1; \ldots; A_k]\!] \Longrightarrow a \in S$
$\vdots$

where $A_1; \ldots; A_k$ are side conditions not involving S.

Proving properties of even numbers

Easy: $4 \in Ev$

$$0 \in Ev \Longrightarrow 2 \in Ev \Longrightarrow 4 \in Ev$$

Trickier: $m \in Ev \Longrightarrow m+m \in Ev$
Idea: induction on the length of the derivation of $m \in Ev$
Better: induction on the *structure* of the derivation
Two cases: $m \in Ev$ is proved by

- rule $0 \in Ev$
  $\Longrightarrow m = 0 \Longrightarrow 0+0 \in Ev$
- rule $n \in Ev \Longrightarrow n+2 \in Ev$
  $\Longrightarrow m = n+2$ and $n+n \in Ev$ (ind. hyp.!)
  $\Longrightarrow m+m = (n+2)+(n+2) = ((n+n)+2)+2 \in Ev$

Rule induction for Ev

To prove

$$n \in Ev \Longrightarrow P\ n$$

by *rule induction* on $n \in Ev$ we must prove

- $P\ 0$
- $P\ n \Longrightarrow P(n+2)$

Rule `Ev.induct`:

$[\![n \in Ev; P\ 0; \bigwedge n.\ P\ n \Longrightarrow P(n+2)]\!] \Longrightarrow P\ n$

An elimination rule

Rule induction in general

Set S is defined inductively.
To prove

$$x \in S \Longrightarrow P\ x$$

by *rule induction* on $x \in S$
we must prove for every rule

$$[\![\ a_1 \in S; \ldots ; a_n \in S\ ]\!] \Longrightarrow a \in S$$

that P is preserved:

$$[\![\ P\ a_1; \ldots ; P\ a_n\ ]\!] \Longrightarrow P\ a$$

In Isabelle/HOL:

apply*(erule S.induct)*

Demo: inductively defined sets

Isabelle/HOL Exercises

1 Counting occurences

Define a function *occurs*, such that *occurs x xs* is the number of occurrences of the element *x* in the list *xs*.

consts *occurs :: "'a ⇒ 'a list ⇒ nat"*

Prove or disprove (by counter example) the lemmas that follow. You may have to prove additional lemmas first. Use the *[simp]*-attribute only if the equation is truly a simplification and is necessary for some later proof.

lemma *"occurs a xs = occurs a (rev xs)"*
lemma *"occurs a xs <= length xs"*

Function *map* applies a function to all elements of a list: *map f* $[x_1,\ldots,x_n]$ = *[f* x_1*,...,f* x_n*]*.

lemma *"occurs a (map f xs) = occurs (f a) xs"*

Function *filter :: ('a ⇒ bool) ⇒ 'a list ⇒ 'a list* is defined by

```
filter P [] = []
filter P (x # xs) = (if P x then x # filter P xs else filter P xs)
```

Find an expression *e* not containing *filter* such that the following becomes a true lemma, and prove it:

lemma *"occurs a (filter P xs) = e"*

With the help of *occurs*, define a function *remDups* that removes all duplicates from a list.

consts *remDups :: "'a list ⇒ 'a list"*

Find an expression *e* not containing *remDups* such that the following becomes a true lemma, and prove it:

lemma *"occurs x (remDups xs) = e"*

With the help of *occurs* define a function *unique*, such that *unique xs* is true iff every element in *xs* occurs only once.

consts *unique :: "'a list ⇒ bool"*

Show that the result of *remDups* is *unique*.

2 Tree traversal

Define a datatype *'a tree* for binary trees. Both leaf and internal nodes store information.

datatype *'a tree =*

Define the functions *preOrder*, *postOrder*, and *inOrder* that traverse an *'a tree* in the respective order.

consts
```
  preOrder :: "'a tree ⇒ 'a list"
  postOrder :: "'a tree ⇒ 'a list"
  inOrder :: "'a tree ⇒ 'a list"
```

Define a function *mirror* that returns the mirror image of an *'a tree*.

consts
```
  mirror :: "'a tree ⇒ 'a tree"
```

Suppose that *xOrder* and *yOrder* are tree traversal functions chosen from *preOrder*, *postOrder*, and *inOrder*. Formulate and prove all valid properties of the form *xOrder (mirror xt) = rev (yOrder xt)*.

Define the functions *root*, *leftmost* and *rightmost*, that return the root, leftmost, and rightmost element respectively.

consts
```
  root :: "'a tree ⇒ 'a"
  leftmost :: "'a tree ⇒ 'a"
  rightmost :: "'a tree ⇒ 'a"
```

Prove or disprove (by counter example) the lemmas that follow. You may have to prove some lemmas first.

lemma *"last(inOrder xt) = rightmost xt"*
lemma *"hd (inOrder xt) = leftmost xt"*

lemma *"hd(preOrder xt) = last(postOrder xt)"*
lemma *"hd(preOrder xt) = root xt"*
lemma *"hd(inOrder xt) = root xt"*
lemma *"last(postOrder xt) = root xt"*

3 Natural deduction

3.1 Propositional logic

The focus of this exercise are single step natural deduction proofs. The following restrictions apply:

- Only the following rules may be used:
  notI: $(A \Longrightarrow False) \Longrightarrow \neg A$,
  notE: $[\![\neg A; A]\!] \Longrightarrow B$,
  conjI: $[\![A; B]\!] \Longrightarrow A \wedge B$,
  conjE: $[\![A \wedge B; [\![A; B]\!] \Longrightarrow C]\!] \Longrightarrow C$,
  disjI1: $A \Longrightarrow A \vee B$,
  disjI2: $A \Longrightarrow B \vee A$,
  disjE: $[\![A \vee B; A \Longrightarrow C; B \Longrightarrow C]\!] \Longrightarrow C$,
  impI: $(A \Longrightarrow B) \Longrightarrow A \longrightarrow B$,
  impE: $[\![A \longrightarrow B; A; B \Longrightarrow C]\!] \Longrightarrow C$,
  mp: $[\![A \longrightarrow B; A]\!] \Longrightarrow B$
  iffI: $[\![A \Longrightarrow B; B \Longrightarrow A]\!] \Longrightarrow A = B$,
  iffE: $[\![A = B; [\![A \longrightarrow B; B \longrightarrow A]\!] \Longrightarrow C]\!] \Longrightarrow C$
  classical: $(\neg A \Longrightarrow A) \Longrightarrow A$
- Only the methods *rule*, *erule* und *assumption* may be used.

lemma *I:* *"A ⟶ A"*
lemma *"(A ∨ B) = (B ∨ A)"*
lemma *"(A ∧ B) ⟶ (A ∨ B)"*
lemma *"((A ∨ B) ∨ C) ⟶ A ∨ (B ∨ C)"*
lemma *K:* *"A ⟶ B ⟶ A"*
lemma *"(A ∨ A) = (A ∧ A)"*
lemma *S:* *"(A ⟶ B ⟶ C) ⟶ (A ⟶ B) ⟶ A ⟶ C"*
lemma *"(A ⟶ B) ⟶ (B ⟶ C) ⟶ A ⟶ C"*
lemma *"¬ ¬ A ⟶ A"*
lemma *"(¬ A ⟶ B) ⟶ (¬ B ⟶ A)"*
lemma *"((A ⟶ B) ⟶ A) ⟶ A"*
lemma *"A ∨ ¬ A"*

3.2 Predicate logic

You may now use the followinh additional rules:

```
exI: P x ⟹ ∃x. P x
exE: ⟦∃x. P x; ⋀x. P x ⟹ Q⟧ ⟹ Q
allI: (⋀x. P x) ⟹ ∀x. P x
allE: ⟦∀x. P x; P x ⟹ R⟧ ⟹ R
```

For each of the following formulae, find a proof or explain why it is not true.

```
lemma "(∀x. P x ⟶ Q) = ((∃x. P x) ⟶ Q)"
lemma "(∀ x. ∀ y. R x y) = (∀ y. ∀ x. R x y)"
lemma "((∃ x. P x) ∨ (∃ x. Q x)) = (∃ x. (P x ∨ Q x))"
lemma "((∀ x. P x) ∨ (∀ x. Q x)) = (∀ x. (P x ∨ Q x))"
lemma "(∀x. ∃y. P x y) ⟶ (∃y. ∀x. P x y)"
lemma "(∃x. ∀y. P x y) ⟶ (∀y. ∃x. P x y)"
lemma "(¬ (∀ x. P x)) = (∃ x. ¬ P x)"
```

3.3 A puzzle

Prove the following proposition with pen and paper, possibly using case distinctions:

If every poor person has a rich father,
then there is a rich person with a rich grandfather.

theorem

```
"∀x. ¬ rich x ⟶ rich (father x) ⟹
∃x. rich (father (father x)) ∧ rich x"
```

Translate your proof into a sequence of Isabelle rule applications. Case distinctions via `case_tac` are allowed.

4 Context-free grammars

This exercise is concerned with context-free grammars (CFGs). Please read section 7.4 in the tutorial which explains how to model CFGs as inductive definitions. Our particular example is about defining valid sequences of parantheses.

4.1 Two grammars

The most natural definition of valid sequences of parantheses is this:

$$S \quad \to \quad \varepsilon \quad | \quad '(' \; S \; ')' \quad | \quad S \; S$$

where ε is the empty word.

A second, somewhat unusual grammar is the following one:

$$T \quad \to \quad \varepsilon \quad | \quad T \; '(' \; T \; ')'$$

Model both grammars as inductive sets S and T and prove $S = T$.

4.2 A recursive function

Instead of a grammar, we can also define valid sequences of paratheses via a test function: traverse the word from left to right while counting how many closing paretheses are still needed. If the counter is 0 at the end, the sequence is valid.

Define this recursive function and prove that a word is in S iff it is accepted by your function. The $\Longrightarrow$ direction is easy, the other direction more complicated.

PART IV:

Advanced Applications and Tools

Yves Bertot:

Coinduction

CoInduction in Coq

Yves Bertot

June 30, 2005

When providing a collection of constructors to define an inductive type, we actually also define a dual operation: a destructor. This destructor is always defined using the same structure of pattern-matching, so that we have a tendency to forget that we do extend the "pattern-matching" capability with a new destructor at each definition.

Constructors and destructors play a dual role in the definition of inductive types. Constructors produce elements of the inductive type, destructors consume elements of the inductive type.

The inductive type itself is defined as the smallest collection of elements that is stable with respect to the constructors: it must contain all constants that are declared to be in the inductive type and all results of the constructors when the arguments of these constructors are already found to be in the inductive type. When considering structural recursion, recursive definitions are functions that consume elements of the inductive type. The discipline of structural recursion imposes that recursive calls *consume* data that is *obtained* through the destructor.

The inductive type uses the constructors and destructors in a specific way. Co-inductive types are the types one obtains when using them in a dual fashion. A co-inductive type will appear as the largest collection of elements that is stable with respect to the destructor. It contains every object that can be destructed by pattern-matching.

The duality goes on when considering the definition of recursive functions. Co-recursive functions are function that produce elements of the co-inductive type. The discipline of guarded co-recursion imposes that co-recursive calls *produce* data that is *consumed* by a constructor. The main practical consequence is that co-inductive types contain objects that look like *infinite objects*.

This rough sketch is more of a philosophical nature. When looking at the details, there are some aspects of co-inductive types that are not so simple to derive from a mere reflection of what happens with inductive types.

The possibility to have co-inductive types in theorem proving tools was studied by Coquand [7], Paulson [19], Leclerc and Paulin-Mohring [16], and Gimenez [13]. Most of these authors were inspired by Aczel [1]. The paper [2] provides a short presentation of terms and (possibly infinite) trees, mainly in set-theoretic terms; it also explains recursion and co-recursion.

In this document, we only consider the use of co-inductive types as it is provided in Coq.

1 Defining a co-inductive types

Since defining a set of constructors automatically defines a destructor, the definition of co-inductive types also relies on the definition of constructors. The same rules of positivity as for inductive types apply. Here are three simple examples of co-inductive types:

```
CoInductive Llist (A:Set) : Set :=
  Lcons : A -> Llist A -> Llist A | Lnil : Llist A.

CoInductive stream (A:Set) : Set :=
  Cons : A -> stream A -> stream A.

CoInductive Ltree (A:Set) : Set :=
  Lnode : A -> Ltree A -> Ltree A -> Ltree A  | Lleaf : Ltree A.
```

As for inductive types, this defines the type and the constructors, it also defines the destructor, so that every element of the co-inductive can be analysed by pattern-matching. However, the definition does not provide an induction principle. The reason for the absence of an induction principle can be explained in two ways, philosophical or technical. Philosophically, the induction principle of an inductive type expresses that this inductive type is minimal (it is a least fixed point), but the co-inductive type is rather viewed as greatest fixed point. Technically, the induction principle actually is a *consumption* tool, that consumes elements of an inductive type to produce proofs in some other type, programed by recursion. However, a co-recursive function can only be used to produce elements of the co-recursive type, so that the only way to deduce anything from an element of a co-inductive type is by pattern-matching.

The type `Llist` given above comes with constructors `Lcons` and `Lnil`. These constructors will make it possible to produce lists, exactly like the lists that we could produce in the inductive type `list`. Here are a few examples.

```
Implicit Arguments Lcons.
Implicit Arguments Cons.

Definition ll123 := Lcons 1 (Lcons 2 (Lcons 3 (Lnil nat))).

Require Import List.

Fixpoint list_to_llist (A:Set) (l:list A)
    {struct l} : Llist A :=
 match l with
   nil => Lnil A
 | a::tl => Lcons a (list_to_llist A tl)
 end.

Definition ll123' := list_to_llist nat (1::2::3::nil).
```

The function `list_to_llist` is recursive and produces elements in a co-inductive type, but we did not make it rely on co-recursion. Rather, it relies on structural recursion as it is provided with the inductive type `list`.

Similar examples which do not rely on co-recursion cannot be provided for the type `stream`, because elements of this type cannot be given with a finite number of uses of the constructor. There is always a need for another element of the co-inductive type. Co-recursion is the solution.

In the coq system, co-recursion is only allowed under a form that can ensure the strong normalization properties that are already satisfied by the inductive part of the calculus. The decision was taken to impose a syntactic criterion: co-recursive values can only appear as argument to constructors and inside branches of pattern-matching constructs. Here is a simple example:

```
CoFixpoint ones : stream nat := Cons 1 ones.
```

This definition underlines a funny aspect of co-recursion: a co-recursive value is not necessarily a function, because it is only constrained to *produce* an element of the co-inductive type. The definition contains a usage of the constructor `Cons` and a reference to the co-recursive value itself, but this co-recursive value is used as an argument to the constructor.

A similar value can be defined in the type `Llist`.

```
CoFixpoint lones : Llist nat := Lcons 1 lones.
```

Clearly, the list that we obtain is not a list that we could have obtained using the function `list_to_llist`. The type `Llist` is "larger" than the type `list`.

Some co-recursive functions can be defined to perform exactly like similar recursive functions on inductive types. Here is an instance:

```
Fixpoint map (A B:Set)(f:A -> B)(l:list A) {struct l} : list B :=
  match l with
    nil => nil
  | a::tl => f a::map A B f tl
  end.

CoFixpoint lmap (A B:Set)(f:A -> B)(l:Llist A) : Llist B :=
  match l with
    Lnil => Lnil B
  | Lcons a tl => Lcons (f a) (lmap A B f tl)
  end.
```

The two functions look similar, but we should bear in mind that the second one can also process infinite lists like `lones`.

2 Computing with co-recursive values

When we manipulate elements of inductive types, we implicitely expect to look at these values in constructor form: constructors applied to other terms in constructor form. However, attempting to put a value like `lones` in constructor form would require an infinity of

unfoldings of its value, this would make the computation non-normalizing. For this reason, a co-recursive value is by default considered to be a normal form. This can be verified by requesting a computation on such a value.

```
Eval simpl in lones.
 = lones : Llist nat
```

However destructing a co-recursive value (with the help of a pattern-matching construct) corresponds to a regular redex and we can check that the first element of `lones` indeed is 1.

```
Eval simpl in
 match lones with Lnil => 0 | Lcons a _ => a end.
 = 1 : nat
```

3 Proving properties of co-inductive values

It is possible to prove that two co-inductive values are equal. The usual approach, identical to what happens in the inductive setting is to show that the two values have the same head constructor applied to the same arguments. However, making the head constructor appear is tricky, because the computation of co-recursive values is usually not performed. One way to provoke this computation is to rely on the following function and theorem.

```
Definition Llist_decompose (A:Set)(l:Llist A) : Llist A :=
  match l with Lnil => Lnil A | Lcons a tl => Lcons a tl end.
Implicit Arguments Llist_decompose.

Theorem Llist_dec_thm :
   forall (A:Set)(l:Llist A), l = Llist_decompose l.
Proof.
 intros A l; case l; simpl; trivial.
Qed.
```

Now, here is an example using this theorem:

```
Theorem lones_cons : lones = Lcons 1 lones.
Proof.
pattern lones at 1; rewrite Llist_dec_thm; simpl.
...
  ============================
   Lcons 1 lones = Lcons 1 lones

trivial.
Qed.
```

This is not a proof by co-recursion, just a proof by pattern-matching.

There are proofs of equality that seem obvious but cannot be performed in the calculus of inductive constructions as it is defined now. This happens when the proofs seems to require some sort of inductive argument. Here is an instance of an impossible proof:

```
Theorem lmap_id : forall (A:Set)(l:Llist A),
   lmap A A (fun x:A => x) l = l.
```

One would like to have an argument of the following form: if the list is nil, then the proof is trivial, if the list is not `nil`, then the head on both sides of the equality are naturally the same, and the equality for the tails should hold by some "inductive" argument. However, there is no induction hypothesis, because there is no inductive list in this statement and the proof of equality can only be proved by using the constructor of equality (because equality is itself an inductive type).

The solution for this kind of problem is to use a co-inductive proposition that expresses the equality of two lists by stating that they have the same elements. Here is the co-inductive definition for this proposition:

```
CoInductive bisimilar (A:Set) : Llist A -> Llist A -> Prop :=
  bisim0 : bisimilar A (Lnil A) (Lnil A)
| bisim1 : forall a l l',
       bisimilar A l l' -> bisimilar A (Lcons a l)(Lcons a l').
```

Proofs that two lists have the same elements can now also be obtained by using co-recursive values, as long as we use the `bisimilar` relation instead of equality. Here is an example of a proof, displayed as term of the calculus of inductive constructions to make the general structure visible. Please note that rewrites using the theorems `eq_ind_r` and `Llist_dec_thm` are performed to introduce the function `Llist_decompose` and force the expansion of the co-recursive function.

```
CoFixpoint lmap_bi (A:Set)(l:Llist A) :
  bisimilar A (lmap A A (fun x:A => x) l) l :=
 @eq_ind_r (Llist A) (Llist_decompose (lmap A A (fun x=> x) l))
   (fun x => bisimilar A x l)
   match l return bisimilar A
                   (Llist_decompose (lmap A A (fun x=>x) l))
                   l with
     Lnil => bisim0 A
   | Lcons a k =>
             bisim1 A a (lmap A A (fun x=> x) k) k (lmap_bi A k)
   end
   (lmap A A (fun x => x) l)
   (Llist_dec_thm A (lmap A A (fun x=>x) l))
.
```

The manual construction of co-inductive proofs is difficult. The alternative approach is to use tactics. The following script performs the same proof, but relying on tactics.

```
Theorem lmap_bi' : forall (A:Set)(l:Llist A),
  bisimilar A (lmap A A (fun x => x) l) l.
```

```
cofix.
intros A l; rewrite (Llist_dec_thm _ (lmap A A (fun x=>x) l)).
case l.
intros a k; simpl.
apply bisim1; apply lmap_bi'.
simpl; apply bisim0.
Qed.
```

The tactic `cofix` is the tactic that declares that the current proof will be a co-recursive value. It introduces a new assumption in the context so that the co-recursive value can be used inside its own definition. However, the same constraints as before exist: the co-recursive value can only be used as input to a constructor. In the case of `lmap_bi`, the use of `lmap_bi'` at the end of the proof is justified by the previous use of the constructor `bisim1`: `lmap_bi'` is thus used to provide an argument to `bisim1`.

In general, the constraint that co-recursive calls are used in correct conditions is only checked at the end of the proof. This sometimes has the unpleasant effect that one believes to have completed a proof and is only rebuked when the `Qed` or `Defined` commands announce that the constructed term is not well-formed. This problem is compounded by the fact that it is hard to control the hypotheses that are used by automatic tactics. Even though we believe the proof of a subgoal should not rely on the co-recursive assumption, it may happen that some tactic like `intuition` uses this assumption in a bad way. One solution to this problem is to use the `clear` tactic to remove the co-recursive assumption before using strong automatic tactics. A second important tool to avoid this problem is a command called `Guarded`, this command can be used at any time during the interactive proofs and it checks whether illegal uses of the co-recursive tactic have already been performed.

4 Applications

Co-inductive types can be used to reason about hardware descriptions [9] concurrent programming [14], finite state automata and infinite traces of execution, and temporal logic [5, 8]. The guarded by constructors structure of co-recursive functions is adapted to representing finite state automata. A few concrete examples are also given in [4].

Co-inductive types are especially well suited to model and reason about lazy functional programs that compute on infinite lists. However, the constraints of having co-recursive calls guarded by constructors imposes that one scrutinizes the structure of recursive functions to understand whether they really can be encoded in the language. One approach, used in [3] is to show that co-inductive objects also satisfy some inductive properties, which make it possible to define functions that have a recursive part, with usual structural recursive calls with respect to these inductive properties, and guarded co-recursive parts.

5 An example: introduction to exact real arithmetics

The work presented in this section is my own, but it is greatly inspired by reading the lecture notes [10] and the thesis [17] and derived papers [18, 15], and by [6]. These papers

should be consulted for further references about exact real arithmetics, lazy computation, and co-inductive types.

We are going to represent real numbers between 0 and 1 (included) as infinite sequences of intervals I_n, where $I_0 = [0, 1]$, $I_{n+1} \subset I_n$ and the size of I_{n+1} is half the size of I_n. Moreover, I_{n+1} will be obtained from I_n in only one of three possible ways:

1. I_{n+1} is the left half of I_n,
2. I_{n+1} is the right half of I_{n+1},
3. I_{n+1} is the center of I_{n+1}: if a and b are the bounds of I_n, then $a + (b - a)/4$ and $a + 3(b - a)/4$ are the bounds of $I_n + 1$.

We can represent any of the intervals I_n using lists of `idigit`, where the type `idigit` is the three element enumerated type containing `L`, `R`, `C`. For instance, the interval [0,1] is given by the empty list, the interval [1/4,3/8] can be represented by the lists `L::C::R::nil`, `L::R::L::nil`, or `C::L::L::nil`. It is fairly easy to write a function of type `list idigit->R` that maps every list to the lower and upper bound of the interval it represents. We are going to represent real numbers by infinite sequences of intervals using the type `stream idigit`.

There is also an easy correspondence from floating-point numbers in binary representation to this representation. Let us first recall what the binary floating-point representation is. Any "binary" floating point is a list of boolean values. Interpreting `true` as the 1 bit and `false` as the 0 bit, a boolean list is interpreted as a real number in the following way:

```
Fixpoint bit_list_to_R (l:list boolean) : Rdefinitions.R :=
  match l with
    nil => 0
  | b::tl => let x := bit_list_to_R tl in
             if b then (1+x)/2 else x/2
  end.
```

We can inject the boolean values into the type `idigit` mapping `true` to `L` and `false` to `R`. It is fairly easy to show that this correspondance can be lifted to lists of booleans and idigits, so that the real number represented by a list is element of the interval represented by the corresponding list.

We represent real numbers by streams of `idigit` elements. The construction relies on associating a sequence of real numbers to each stream (actually the lower bounds of the intervals) and to show that this sequence converges to a limit. To ease our reasoning, we will also describe the relation between a stream and a real value using a co-inductive property:

```
CoInductive represents : stream idigit -> Rdefinitions.R -> Prop :=
  reprL : forall s r, represents s r -> (0 <= r <= 1)%R ->
           represents (Cons L s) (r/2)
| reprR : forall s r, represents s r -> (0 <= r <= 1)%R ->
           represents (Cons R s) ((r+1)/2)
| reprC : forall s r, represents s r -> (0 <= r <= 1)%R ->
           represents (Cons C s) ((2*r+1)/4).
```

We could also use infinite lists of booleans to represent real numbers. This is the usual representation of numbers. This representation also corresponds to sequences of intervals, but it has bad programming properties. In this representation, if we know that a number is very close to 1/2 but we don't know whether it is larger or smaller, we cannot produce the first bit. For instance, the number 1/3 is represented by the infinite sequence .0101... and the number 1/6 is represented by the infinite sequence .0010101... Adding the two numbers should yield the number 1/2. However, every finite prefix of .010101... represents an interval that contains numbers that are larger than 1/3 and numbers that are smaller than 1/3. Similarly, every finite prefix of .0010101... contains a numbers that are larger than 1/6 and numbers that are smaller. By only looking at a finite prefix of both numbers, we cannot decide whether the first bit of the result should be a 0 or a 1, because no number larger than 1/2 can be represented by a sequence starting with a 0 and no number smaller than 1/2 can be represented by a sequence starting with a 1.

With the extra digit, `C`, we can perform the computation as follows:

1. having observed that the first number has the form $x = LRx'$, we know that this number is between 1/4 and 1/2,
2. having observed that the second number has the form $y = LLy'$, we know that this number is between 0 and 1/4,
3. we know that the sum is between 1/4 and 3/4. therefore, we know that the sum is an element of the interval represented by `C::nil`, and we can output this digit.

We can also go on to output the following digits. In usual binary representation, if v is the number represented by the sequence s, then the number represented by the sequence $0s$ is $v/2$ and the number represented by the sequence $1s$ is $(v+1)/2$. This interpretation carries over to the digits `L` and `R`, respectively. For the digit `C`, we know that the sequence `C`s represents $(2v+1)/4$. Thus, if we come back to the computation of $1/3 + 1/6$, we know that x' is $4 * x - 1$, y' is $4 * y$, and the result should have the form `C::`z, where z is the representation of $(x' + y' + 1)/4$ (since $x' + y' + 1)/4$ is 1/2, we see that the result of the sum is going to be an infinite sequence of `C` digits.

We are now going to provide a few functions on streams. As a first example, the function `rat_to_stream` maps any two integers a b to a stream. When a/b is between 0 and 1, the result stream is the representation of this rational number.

```
CoFixpoint rat_to_stream (a b:Z) : stream idigit :=
  if Z_le_gt_dec (2*a) b then
    Cons L (rat_to_stream (2*a) b)
  else
    Cons R (rat_to_stream (2*a-b) b)
```

For the second example, we compute an affine combination of two numbers with rational coefficients. We will define the function that constructs the representation of the following formula.

$$\frac{a}{a'}v_1 + \frac{b}{b'}v_2 + \frac{c}{c'}$$

The numbers a, a', ... are positive integers and a', b', and c' are non-zero (this sign restriction only serves to make the example shorter).

We choose to define a one-argument function, where the argument is a record holding all the values a, a', ..., v_1, v_2. We define a type for this record and a predicate to express the sign conditions.

```
Record affine_data : Set :=
 {m_a : Z; m_a' : Z; m_b : Z; m_b' : Z; m_c : Z; m_c' : Z;
  m_v1 : stream idigit; m_v2 : stream idigit}.

Definition positive_coefficients (x:affine_data) :=
  0 <= m_a x /\ 0 < m_a' x /\ 0 <= m_b x /\ 0 < m_b' x
  /\ 0 <= m_c x /\ 0 < m_c' x.
```

We define a function `axbyc` of type

```
 forall x, positive_coefficients x -> stream idigit.
```

The algorithm contains two categories of computing steps. In computing steps of the first category, a digit of type `idigit` is produced, because analysing the values of a, a', ... makes it possible to infer that the result will be in a precise part of the interval. The result then takes the form

$$\texttt{Cons}\ d\ (\texttt{axbyc}\ \langle a_1, a'_1, b_1, b'_1, c_1, c'_1, v_1, v_2\rangle)$$

Where d is a digit and the values of a_1, a'_1, ... depend on the digit.

1. if $c/c' \geq 1/2$, then the result is sure to be in the right part of the interval, the digit d is `R` and the new parameters are chosen so that $a_1/a'_1 = 2a/a'$, $b_1/b'_1 = 2b/b'$, $c_1/c'_1 = (2c - c')/c'$, because of the following equality:

$$\frac{1}{2}(\frac{2a}{a'}v_1 + \frac{2b}{b'}v_2 + \frac{2c - c'}{c'}) + \frac{1}{2} = \frac{a}{a'}v_1 + \frac{b}{b'}v_2 + \frac{c}{c'}$$

2. if $2(ab'c' + ba'c' + a'b'c) \leq a'b'c'$, then the result is sure to be in the left part of the interval, the digit d is `L` and the new parameters are chosen so that $a_1/a'_1 = 2a/a'$, $b_1/b'_1 = 2b/b'$, $c_1/c'_1 = 2c/c'$ (we do not detail the justification),

3. if $(4(ab'c' + ba'c' + a'b'c) \leq 3a'b'c'$ and $4 * c \geq c'$, then the result is sure to belong to the center sub-interval, the digit d is `C` and the new parameters are chosen so that $a_1/a'_1 = 2a/a'$, $b_1/b'_1 = 2b/b'$, $c_1/c'_1 = (4c - c')/2c'$.

The various cases of these productive steps are described using the following functions:

```
Definition prod_R x :=
  Build_affine_data (2*m_a x) (m_a' x) (2*m_b x) (m_b' x)
  (2*m_c x - m_c' x) (m_c' x) (m_v1 x) (m_v2 x).
```

```
Definition prod_L x :=
  Build_affine_data (2*m_a x) (m_a' x) (2*m_b x) (m_b' x)
  (2*m_c x) (m_c' x) (m_v1 x) (m_v2 x).

Definition prod_C x :=
  Build_affine_data (2*m_a x) (m_a' x) (2*m_b x) (m_b' x)
  (4*m_c x - m_c' x) (2*m_c' x) (m_v1 x) (m_v2 x).
```

In the second category of computing steps the values v_1 and v_2 are scrutinized, so that the interval for the potential values of the result is reduced as one learns more information about the inputs. If the values v_1 and v_2 have the form `Cons` d_1 v_1' and `Cons` d_2 v_2' respectively, The result then takes the form

$$\texttt{axbyc}\ \langle a, 2a', b, 2b', c_1, c_1', v_1', v_2' \rangle$$

Only the parameters c_1 and c_1' take a different form depending on the values of d_1 and d_2. The correspondance is given in the following table.

d_1	d_2	c_1	c_1'
L	L	c	c'
L	R	$bc' + 2cb'$	$2b'c'$
R	L	$ac' + 2ca'$	$2a'c'$
L	C	$bc' + 4cb'$	$4b'c'$
C	L	$ac' + 4ca'$	$4a'c'$
R	C	$2ba'c' + ab'c' + 4cb'a'$	$4a'b'c'$
C	R	$2ba'c' + ab'c' + 4cb'a'$	$4b'a'c'$
R	R	$ab'c' + ba'c' + 2ca'b'$	$2a'b'c'$
C	C	$ba'c' + ab'c' + 4cb'a'$	$4b'a'c'$

For justification, let us look only at the case where $v_1 = \texttt{R}v_1'$ and $v_2 = \texttt{C}v_2'$. In this case we have the following equations:

$$\begin{aligned} \frac{a}{a'}v_1 + \frac{b}{b'}v_2 + \frac{c}{c'} &= \frac{a}{a'}(\frac{1}{2}v_1' + \frac{1}{2}) + \frac{b}{b'}(\frac{1}{2}v_2' + \frac{1}{4}) + \frac{c}{c'} \\ &= \frac{a}{2a'}v_1' + \frac{b}{2b'}v_2' + \frac{2ba'c' + ab'c' + 4cb'a'}{4a'b'c'} \end{aligned}$$

This category of computation is taken care of by a function with the following form:

```
Definition axbyc_consume (x:affine_data) :=
 let (a,a',b,b',c,c',v1,v2) := x in
 let (d1,v1') := v1 in let (d2,v2') := v2 in
 let (c1,c1') :=
  match d1,d2 with
  | L,L => (c, c')
  | L,R => (b*c'+2*c*b', 2*b'*c')
  | R,L => (a*c'+2*c*a', 2*a'*c')
```

```
  | L,C => (b*c'+4*c*b', 4*b'*c')
  | C,L => (a*c'+4*c*a', 4*a'*c')
  | R,C => (2*a*b'*c'+b*a'*c'+4*c*a'*b', 4*a'*b'*c')
  | C,R => (2*b*a'*c'+a*b'*c'+4*c*b'*a', 4*b'*a'*c')
  | R,R => (a*b'*c'+b*a'*c'+2*c*a'*b', 2*a'*b'*c')
  | C,C => (b*a'*c'+a*b'*c'+4*c*b'*a', 4*b'*a'*c')
  end in
 Build_affine_data a (2*a') b (2*b') c1 c1' v1' v2'.
```

From the point of view of co-recursive programming, the first category of computing steps gives regular guarded-by-constructor corecursive calls. The second category of computing steps does not give any guarded corecursion. We need to separate the second category in an auxiliary function. We choose to define this auxiliary function by well-founded induction. The recursive function performs the various tests with the help of an auxiliary test function:

```
  Parameter axbyc_test :
  forall x,
   positive_coefficients x ->
   m_c' x <= 2*m_c x+
   2*(m_a x*m_b' x*m_c' x +
        m_b x*m_a' x*m_c' x + m_a' x*m_b' x*m_c x)<=
     m_a' x*m_b' x*m_c' x+
   4*(m_a x*m_b' x*m_c' x +
        m_b x*m_a' x*m_c' x + m_a' x*m_b' x*m_c x)<=
     3*m_a' x*m_b' x*m_c' x /\ m_c' x <= 4*m_c x+
   m_a' x < 8*m_a x  m_b' x < 8*m_b x.
```

In the first three cases, the recursive function just returns the value that it received, together with the proofs of the properties. To carry these agregates of values and proofs, we defined a specific type to combine these values and proofs.

```
Inductive decision_data : Set :=
  caseR : forall x:affine_data, positive_coefficients x ->
          m_c' x <= 2*m_c x -> decision_data
| caseL : forall x:affine_data, positive_coefficients x ->
          2*(m_a x*m_b' x*m_c' x +
             m_b x*m_a' x*m_c' x + m_a' x*m_b' x*m_c x)<=
           m_a' x*m_b' x*m_c' x -> decision_data
| caseC : forall x:affine_data, positive_coefficients x ->
          4*(m_a x*m_b' x*m_c' x +
             m_b x*m_a' x*m_c' x + m_a' x*m_b' x*m_c x)<=
          3*m_a' x*m_b' x*m_c' x -> m_c' x <= 4*m_c x ->
          decision_data.
```

The recursive function will thus have the type

```
 forall x, positive_coefficient x -> decision_data.
```

The definition has the following form:

```
Definition axbyc_rec_aux (x:affine_data)
   : (forall y, order y x ->
         positive_coefficients y -> decision_data)->
     positive_coefficients x -> decision_data :=
  fun f Hp =>
  match A.axbyc_test x Hp with
    inleft (inleft (left H)) => caseR x Hp H
  | inleft (inleft (right H)) => caseL x Hp H
  | inleft (inright (conj H1 H2)) => caseC x Hp H1 H2
  | inright H =>
    f (axbyc_consume x)
      (A.axbyc_consume_decrease x Hp H)
      (A.axbyc_consume_pos x Hp)
  end.

Definition axbyc_rec :=
  well_founded_induction A.order_wf
  (fun x => positive_coefficients x -> decision_data)
  axbyc_rec_aux.
```

The definition of `axbyc_rec` of course relies on proofs to ensure that `axbyc_consume` preserves the sign conditions and make the measure decrease, we do not include these proofs in these notes.

The main co-recursive function relies on the auxiliary recursive function to perform all the recursive calls that are not productive, the value returned by the auxiliary function is suited to produce data and co-recursive calls are then allowed.

```
CoFixpoint axbyc (x:affine_data)
   (h:positive_coefficients x):stream idigit :=
  match axbyc_rec x h with
    caseR y Hpos H => Cons R (axbyc (prod_R y) (A.prod_R_pos y Hpos H))
  | caseL y Hpos H => Cons L (axbyc (prod_L y) (A.prod_L_pos y Hpos))
  | caseC y Hpos H1 H2 =>
        Cons C (axbyc (prod_C y) (A.prod_C_pos y Hpos H2))
  end.
```

This function relies on auxiliary functions to perform the relevant updates of the various coefficients. For instance, here is the function `prod_C`:

```
Definition prod_C x :=
  Build_affine_data (2*m_a x) (m_a' x) (2*m_b x) (m_b' x)
  (4*m_c x-m_c' x) (m_c' x) (m_v1 x) (m_v2 x).
```

For each of these functions, it is also necessary to prove that they preserve the sign conditions, these proofs are fairly trivial.

It requires more work to prove that the function is correct, in the sense that it does produce the representation of the right real number, but this proof is too long to fit in these short tutorial notes. More work is also required to make the function more efficient, for instance by dividing `a` (resp. `b`, `c`) and `a'` (resp. `b'`, `c'`) by they greatest common divisor at each step.

The representation for real numbers proposed in [10] is very close to the representation used in these notes, except that the initial interval is [-1,1], and the three digits are interpreted as the sub-intervals [-1,0], [0,-1], [-1/2,1/2]. The whole set of real numbers is then encoded by multiplying a number in [-1,1] by an exponent of 2 (as in usual scientific, floating point notation). The work presented in [17] shows that both the representation in these notes and the representation in [10] are a particular case of a general framework based on overlapping intervals and proposes a few other solutions. In these notes, we have decided to restrict ourselves to affine binary operations, which makes it possible to obtain addition and multiplication by a rational number, but the most general setting relies on homographic and quadratic functions, which make it possible to obtain addition, multiplication, and division, all in one shot.

The method of separating a recursive part from a co-recursive part in a function definition was already present in [3]. However, the example of [3] is more complex because the functions are *partial*: there are streams for which eventual productivity is not ensured and a stronger description technique is required. This stronger technique is described in [4] as *ad-hoc* recursion. The papers [11, 12] propose an alternative foundation to functions that mix recursive and co-recursive parts.

6 Exercises

increasing streams Define a co-inductive predicate that is satisfied by any stream such that, if n and m are consecutive elements, then $n \leq m$.

Fibonnacci streams Define a co-inductive predicate, called `local_fib`, that is satisfied by any stream such that, if n, m, p are consecutive elements, then $p = n + m$. Define a co-recursive function that constructs a fibonacci stream whose first two elements are 1. Prove that the stream that is created satisfies the two predicates (`increasing` and `local_fib`).

7 Solutions

```
Require Export Omega.

CoInductive increasing : stream nat -> Prop :=
  ci : forall a b tl, a <= b -> increasing (Cons b tl) ->
              increasing (Cons a (Cons b tl)).

CoInductive local_fib : stream nat -> Prop :=
  clf : forall a b tl, local_fib (Cons b (Cons (a+b) tl)) ->
       local_fib (Cons a (Cons b (Cons (a+b) tl))).

CoFixpoint fibo_str (a b:nat) : stream nat := Cons a (fibo_str b (a + b)).

Definition str_decompose (A:Set)(s:stream A) : stream A :=
  match s with Cons a tl => Cons a tl end.

Implicit Arguments str_decompose.

Theorem str_dec_thm : forall (A:Set)(s:stream A), str_decompose s = s.
Proof.
intros A [a tl];reflexivity.
Qed.

Implicit Arguments str_dec_thm.

Theorem increasing_fibo_str :
  forall a b, a <= b -> increasing (fibo_str a b).
Proof.
Cofix.
intros a b Hle.
rewrite <- (str_dec_thm (fibo_str a b));simpl
assert (Heq:(fibo_str b (a+b))=(Cons b (fibo_str (a+b) (b+(a+b))))).
rewrite <- (str_dec_thm (fibo_str b (a+b)));simpl;auto.
rewrite Heq.
constructor.
assumption.
rewrite <- Heq.
apply increasing_fibo_str.
omega.
Qed.

Theorem increasing_fib : increasing (fibo_str 1 1).
Proof.
```

```
 apply increasing_fibo_str;omega.
Qed.

Theorem local_fib_str :
  forall a b, local_fib (fibo_str a b).
Proof.
cofix.
intros a b.
assert (Heq :
          (fibo_str b (a+b)) =
          (Cons b (Cons (a+b)(fibo_str (b+(a+b))((a+b)+(b+(a+b))))))).
rewrite <- (str_dec_thm (fibo_str b (a+b))); simpl.
rewrite <- (str_dec_thm (fibo_str (a+b) (b+(a+b)))); simpl;auto.
rewrite <- (str_dec_thm (fibo_str a b)); simpl.
rewrite Heq.
constructor.
rewrite <- Heq.
apply local_fib_str.
Qed.

Theorem local_fib_fib : local_fib (fibo_str 1 1).
Proof.
 apply local_fib_str.
Qed.
```

References

[1] Peter Aczel. *Non-Well-Founded Sets*. CSLI Lecture Notes, volume 14, 1988.

[2] Yves Bertot. Algebras and Coalgebras. *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *Lecture Notes in Computer Science*, Springer-Verlag, 2002.

[3] Yves Bertot. Filters on CoInductive Streams, an Application to Eratosthenes' Sieve. *Typed Lamdba-Calculi and Applications'05*, volume 3461 of *Lecture Notes in Computer Science*, Springer-Verlag, 2005.

[4] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art:the Calculus of Inductive Constructions*. Springer-Verlag, 2004.

[5] Pierre Castéran and Davy Rouillard. Reasoning about parametrized automata. In *Proceedings, 8-th International Conference on Real-Time System*, volume 8, pages 107–119, 2000.

[6] Alberto Ciaffaglione and Pietro Di Gianantonio. A Co-Inductive Approach to Real Numbers. *Types for Proofs and Programs*, volume 1956 of *Lecture Notes in Computer Science*, Springer-Verlag, 2000.

[7] Thierry Coquand. Infinite objects in Type Theory. *Types for Proofs and Programs*, volume 806 of *Lecture Notes in Computer Science*, Springer-Verlag, 1993.

[8] Solange Coupet-Grimal. An axiomatization of linear temporal logic in the calculus of inductive constructions. *Journal of Logic and Computation*, 13(6):801–813, 2003.

[9] Solange Coupet-Grimal and Line Jakubiec. Hardware verification using co-induction in coq. In *TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[10] Abbas Edalat and Reinhold Heckmann. Computing with Real Numbers. *Applied semantics*, volume 2395 of *Lecture Notes in Computer Science*, Springer-Verlag, 2002.

[11] Pietro di Gianantonio and Marino Miculan. A unifying approach to recursive and co-recursive definitions. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs*, volume 2646 of *LNCS*, pages 148–161. Springer Verlag, 2003.

[12] Pietro di Gianantonio and Marino Miculan. Unifying recursive and co-recursive definitions in sheaf categories. In Igor Walukiewicz, editor, *Foundations of Software Science and Computation Structures (FOSSACS'04)*, volume 2987 of *LNCS*. Springer Verlag, 2004.

[13] Eduardo Giménez. Codifying guarded definitions with recursive schemes. *Types for Proofs and Programs*, volume 996 of *Lecture Notes in Computer Science*, Springer-Verlag, 1994.

[14] Eduardo Giménez. An Application of Co-Inductive Types in Coq: Verification of the Alternating Bit Protocol. *Types for Proofs and Programs*, volume 1158 of *Lecture Notes in Computer Science*, Springer-Verlag, 1995.

[15] Jesse Hughes and Milad Niqui. Admissible digit sets. To appear in *Theoretical Computer Science*, special issue on real numbers and computers, 2005.

[16] François Leclerc and Christine Paulin-Mohring. Programming with streams in coq. A case study: the sieve of Eratosthenes. *Types for Proofs and Progams*, volume 806 of *Lecture Notes in Computer Science*, Springer-Verlag, 1993.

[17] Milad Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Raadboud University, Nijmegen, 2004.

[18] Milad Niqui. Formalising Exact Arithmetic in Type Theory. First conference on computability in Europe, *CiE2005*, volume 3526 of *Lecture Notes in Computer Science*, Springer-Verlag, 2005.

[19] Lawrence C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. *Conference on Automated Deduction*, volume 814 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag 1994.

Jean-Christophe Filliatre:

WHY system and Proofs about programs

TYPES summer school 2005

Program Verification using Coq
Introduction to the WHY tool

Jean-Christophe Filliâtre

CNRS – Université Paris Sud

Contents

Introduction

These lecture notes present some techniques to verify programs correctness using the Coq proof assistant.

In Chapter 1, we show how to use Coq directly to verify purely functional programs, using the Coq extraction mechanism which produces correct ML programs out of constructive proofs. This chapter describes techniques to define and prove correct functions in Coq, focusing on issues such as partial functions, non-structurally recursive functions, or the use of dependent types to specify functions. The interested reader will find much more material related to this subject in Y. Bertot and P. Castéran's book dedicated to Coq [3].

In Chapter 2, we introduce the Why tool for the verification of imperative programs. The Why tool is actually not specifically related to Coq, since it offers a wide range of other back-end provers (PVS, Isabelle/HOL, Simplify, etc.). But Coq plays a particular role since Why can interpret imperative programs as purely functional programs in Coq, providing increased trust in the verification process and making a clear connection between the two chapters of these notes. The interested reader will find more material related to Why on its web site `http://why.lri.fr/`, including a reference manual, many examples and links to related tools for the verification of C and Java programs.

Chapter 1

Verification of purely functional programs

In this chapter, we focus on the specification and verification of purely functional programs. We show how the Coq proof assistant can be used to produce correct ML code. This chapter is illustrated using the case study of balanced binary search trees, which constitutes an example of purely functional program simultaneously useful and complex.

In the following, we call *informative* what lies in the sort `Set` and *logic* what lies in the sort `Prop`. This sort distinction is exploited by the Coq *extraction* mechanism [19, 20, 17, 18]. This tool extracts the informative contents of a Coq term as an ML program. The logical parts disappear (or subsist as a degenerated value with no associated computation). The theoretical foundations of program extraction can be found in the references above.

1.1 Immediate method

The most immediate way to verify a purely functional program consists in defining it as a Coq function and then to prove some properties of this function. Indeed, most (purely functional) ML programs can be written in the Calculus of Inductive Constructions.

Generally speaking, we first define in Coq a "pure" function, that is with a purely informative type a la ML (a type from system F). Let us assume here a function we only one argument:

$$f \quad : \quad \tau_1 \to \tau_2$$

We show that this function realizes a given specification $S : \tau_1 \to \tau_2 \to \texttt{Prop}$ with a theorem of the shape

$$\forall x.\ (S\ x\ (f\ x))$$

The proof is conducted following the definition of f.

Example. We use a finite sets library based on balanced binary trees as a running example. We first introduce a datatype of binary trees containing integers

```
Inductive tree : Set :=
| Empty
| Node : tree -> Z -> tree -> tree.
```

and a membership relation `In` stating that an element occurs in a tree (independently of any insertion choice):

```
Inductive In (x:Z) : tree -> Prop :=
| In_left : forall l r y, (In x l) -> (In x (Node l y r))
| In_right : forall l r y, (In x r) -> (In x (Node l y r))
| Is_root : forall l r, (In x (Node l x r)).
```

A function testing for the empty set can be defined as

```
Definition is_empty (s:tree) : bool := match s with
| Empty => true
| _ => false end.
```

and its correctness proof is stated as

```
Theorem is_empty_correct :
  forall s, (is_empty s)=true <-> (forall x, ~(In x s)).
```

The proof follows the definition of `is_empty` and is only three lines long:

```
Proof.
  destruct s; simpl; intuition.
  inversion_clear H0.
  elim H with z; auto.
Qed.
```

Let us consider now the membership test within a binary search tree. We first assume an ordering relation over integers:

```
Inductive order : Set := Lt | Eq | Gt.
Hypothesis compare : Z -> Z -> order.
```

Then we define a function `mem` looking for an element in a tree which is *assumed* to be a binary search tree:

```
Fixpoint mem (x:Z) (s:tree) {struct s} : bool := match s with
| Empty =>
   false
| Node l y r => match compare x y with
   | Lt => mem x l
   | Eq => true
   | Gt => mem x r
  end
end.
```

The correctness proof of this function requires the definition of a binary search tree, as the following `bst` predicate:

```
Inductive bst : tree -> Prop :=
| bst_empty :
   (bst Empty)
| bst_node :
   forall x (l r : tree),
   bst l -> bst r ->
   (forall y, In y l -> y < x) ->
   (forall y, In y r -> x < y) -> bst (Node l x r).
```

Now the correctness of the `mem` function can be stated:

```
Theorem mem_correct :
  forall x s, (bst s) -> (mem x s=true <-> In x s).
```

We see on this example that the specification S has the particular shape $P\ x \to Q\ x\ (f\ x)$. P is called a *precondition* and Q a *postcondition*.

Modularity. When trying to prove `mem_correct` we start with `induction s; simpl` to follow `mem`'s definition. The first case (`Empty`) is trivial. On the second one (`Node s1 z s2`) we bump into the term `match compare x z with ...` and it is not possible to go further. Indeed, we know nothing about the `compare` function used in `mem`. We have to specify it first, using for instance the axiom

```
Hypothesis compare_spec :
  forall x y, match compare x y with
  | Lt => x<y
  | Eq => x=y
  | Gt => x>y
  end.
```

Then we can use this specification in the following way:

```
 generalize (compare_spec x z); destruct (compare x z).
```

The proof is completed without difficulty.

Note. For purely informative functions such as `is_empty` or `mem`, the extracted program is identical to the `Coq` term. As an example, the command `Extraction mem` gives the following ocaml code:

```
let rec mem x = function
  | Empty -> false
  | Node (l, y, r) ->
    (match compare x y with
       | Lt -> mem x l
       | Eq -> true
       | Gt -> mem x r)
```

1.1.1 The case of partial functions

A first difficulty occurs when the function is partial, i.e. has a Coq type of the shape

$$f : \forall x : \tau_1.\ (P\ x) \to \tau_2$$

The typical case is a division function expecting a non-zero divisor.

In our running example, we may want to define a function `min_elt` returning the smallest element of a set which is *assumed* to be non-empty (that is the leftmost element in the binary search tree). We can give this function the following type:

$$\texttt{min\_elt} : \forall s : \texttt{tree}.\ \neg s = \texttt{Empty} \to \texttt{Z} \tag{1.1}$$

where the precondition is $\neg s = \texttt{Empty}$. The specification of `min_elt` can be stated as

$$\forall s.\ \forall h : \neg s = \texttt{Empty}.\ \texttt{bst}\ s \to \texttt{In}\ (\texttt{min\_elt}\ s\ h)\ s \wedge \forall x.\ \texttt{In}\ x\ s \to \texttt{min\_elt}\ s\ h \leq x$$

with the same precondition as the function itself (hypothesis h) together with the other precondition `bst` s. The hypothesis h is mandatory to be able to apply `min_elt`. We see that using a partial function in Coq is not easy: one has to pass proofs as arguments, and proof terms may be difficult to build.

Even the definition of a partial function may be difficult. Let us write a function `min_elt` with type (1.1). The ML code we have in mind is:

```
let rec min_elt = function
  | Empty -> assert false
  | Node (Empty, x, _) -> x
  | Node (l, _, _) -> min_elt l
```

Unfortunately, the Coq definition is more difficult. First, the `assert false` in the first case of the pattern matching corresponds to an absurd case (we assumed a non-empty tree). The Coq definition expresses this absurdity using the `False_rec` elimination applied to a proof of `False`. So one has to build such a proof from the precondition. Similarly, the third case of the pattern matching makes a recursive call to `min_elt` and thus we have to build a proof that `l` is non-empty. Here it is a consequence of the pattern matching which has already eliminated the case where `l` is `Empty`. In both cases, the necessity to build these proof terms complicates the pattern matching, which must be dependent. We get the following definition:

```
Fixpoint min_elt (s:tree) (h:~s=Empty) { struct s } : Z :=
  match s return ~s=Empty -> Z with
  | Empty =>
    (fun h => False_rec _ (h (refl_equal Empty)))
  | Node l x _ =>
    (fun h => match l as a return a=l -> Z with
              | Empty => (fun _ => x)
              | _ => (fun h => min_elt l
                                 (Node_not_empty _ _ _ _ h))
              end (refl_equal l))
  end h.
```

The first proof (argument of `False_rec`) is built directly. The second one uses the following lemma:

```
Lemma Node_not_empty : forall l x r s, Node l x r=s -> ~s=Empty.
```

We can now prove the correctness of `min_elt`:

```
Theorem min_elt_correct :
  forall s (h:~s=Empty), bst s ->
    In (min_elt s h) s /\
    forall x, In x s -> min_elt s h <= x.
```

Once again, the proof is conducted following the definition of the function and does not contain any difficulty.

One can check that the extracted code is indeed the one we had in mind. `Extraction min_elt` outputs:

```
let rec min_elt = function
  | Empty -> assert false (* absurd case *)
  | Node (l, x, t) ->
      (match l with
         | Empty -> x
         | Node (t0, z0, t1) -> min_elt l)
```

There are several points of interest here. First, the use of `False_rec` is extracted to `assert false`, which is precisely the expected behavior (we proved that this program point was not reachable, so it is legitimate to say that reaching it is absurd i.e. a "proof" of `false`). Second, we see that the logical arguments that were complicating the definition have disappeared in the extracted code (since they were in sort `Prop`).

Another solution is to define the `min_elt` function using a proof rather than a definition. It is then easier to build the proof terms (using the interactive proof editor). Here the definition-proof is rather simple:

```
Definition min_elt : forall s, ~s=Empty -> Z.
Proof.
  induction s; intro h.
  elim h; auto.
  destruct s1.
  exact z.
  apply IHs1; discriminate.
Defined.
```

But it is more difficult to be convinced that we are building the right function (as long as we haven't proved its correctness). In particular, one has to use the automatic tactics such as `auto` with great care, since it could build a function different from the one we have in mind. One the example above, `auto` is only used on a logical goal (`Empty=Empty`).

One way to get convinced that the underlying code is the right one is to have a look at the extracted code. Here we get exactly the same as before.

The `refine` tactic helps in defining partial functions (but not only). It allows the user to give an incomplete proof term, some parts being omitted (when denoted by _) and turned into subgoals. We can redefine the `min_elt` function using the `refine` tactic as follows:

```
Definition min_elt : forall s, ~s=Empty -> Z.
refine
  (fix min (s:tree) (h:~s=Empty) { struct s } : Z :=
  match s return ~s=Empty -> Z with
  | Empty =>
      (fun h => _)
  | Node l x _ =>
      (fun h => match l as a return a=l -> Z with
                | Empty => (fun _ => x)
                | _ => (fun h => min_elt l _)
                end _)
  end h).
```

We get two subgoals that are easy to discharge. However, we notice that a dependent matching is still required.

A last solution consists in making the function total, by completing it in an arbitrary way out of its definition domain. Here we may choose to return the value 0 when the set is empty. This way we avoid the logical argument $\neg s = \texttt{Empty}$ and all its nasty consequences. The type of `min_elt` is back to `tree` $\rightarrow$ `Z` and its definition quite simple:

```
Fixpoint min_elt (s:tree) : Z := match s with
  | Empty => 0
  | Node Empty z _ => z
  | Node l _ _ => min_elt l
end.
```

The correctness theorem is still the same, however:

```
Theorem min_elt_correct :
  forall s, ~s=Empty -> bst s ->
    In (min_elt s) s /\
    forall x, In x s -> min_elt s <= x.
```

The correctness statement still has the precondition $\neg s = \texttt{Empty}$, otherwise it would not be possible to ensure In (`min_elt` s) s.

Note. The division function `Zdiv` over integers is defined this way as a total function but its properties are only provided under the assumption that the divisor is non-zero.

Note. Another way to make the function `min_elt` total would be to make it return a value of type `option Z`, that is `None` when the set is empty and `Some` m when a smallest element m exists. But then the underlying code is slightly different (and so is the correctness statement).

1.1.2 Functions that are not structurally recursive

Issues related to the definition and the use of a partial function are similar to the ones encountered when defining and proving correct a recursive function which is not structurally recursive.

Indeed, one way to define such a function is to use a principle of well-founded induction, such as

```
well_founded_induction
   : forall (A : Set) (R : A -> A -> Prop),
     well_founded R ->
     forall P : A -> Set,
     (forall x : A, (forall y : A, R y x -> P y) -> P x) ->
     forall a : A, P a
```

But then the definition requires to build proofs of `R y x` for each recursive call on `y`; we are faced to the same problems, but also to the same solutions, mentioned in the section above.

Let us assume we want to define a function `subset` checking for set inclusion on our binary search trees. A possible ML code is the following:

```
let rec subset s1 s2 = match (s1, s2) with
  | Empty, _ ->
      true
  | _, Empty ->
      false
  | Node (l1, v1, r1), (Node (l2, v2, r2) as t2) ->
      let c = compare v1 v2 in
      if c = 0 then
        subset l1 l2 && subset r1 r2
      else if c < 0 then
        subset (Node (l1, v1, Empty)) l2 && subset r1 t2
      else
        subset (Node (Empty, v1, r1)) r2 && subset l1 t2
```

We see that recursive calls are performed on trees that are not always strict sub-terms of the initial arguments (not mentioning the additional difficulty of a simultaneous recursion on two arguments). Though there exists a simple termination criterion, that is the total number of elements in the two trees.

Thus we first establish a well-founded induction principle over two trees based on the sum of their cardinalities:

```
Fixpoint cardinal_tree (s:tree) : nat := match s with
  | Empty => 0
  | Node l _ r => (S (plus (cardinal_tree l) (cardinal_tree r)))
end.

Lemma cardinal_rec2 :
```

```
forall (P:tree->tree->Set),
(forall (x x':tree),
   (forall (y y':tree),
     (lt (plus (cardinal_tree y) (cardinal_tree y'))
         (plus (cardinal_tree x) (cardinal_tree x'))) -> (P y y'))
   -> (P x x')) ->
forall (x x':tree), (P x x').
```

The proof is simple: we first manage to reuse a well-founded induction principle over type `nat` provided in the Coq library, namely `well_founded_induction_type_2`, and then we prove that the relation is well-founded since it is of the shape `lt` $(f\ y\ y')$ $(f\ x\ x')$ and because `lt` itself is a well-founded relation over `nat` (another result from the Coq library):

```
Proof.
  intros P H x x'.
  apply well_founded_induction_type_2 with
    (R:=fun (yy' xx':tree*tree) =>
        (lt (plus (cardinal_tree (fst yy')) (cardinal_tree (snd yy')))
            (plus (cardinal_tree (fst xx')) (cardinal_tree (snd xx'))))));
    auto.
    apply (Wf_nat.well_founded_ltof _
            (fun (xx':tree*tree) =>
              (plus (cardinal_tree (fst xx')) (cardinal_tree (snd xx'))))).
Save.
```

We are now in position to define the `subset` function with a definition-proof using the `refine` tactic:

```
Definition subset : tree -> tree -> bool.
Proof.
```

First we apply the induction principle `cardinal_rec2`:

```
intros s1 s2; pattern s1, s2; apply cardinal_rec2.
```

Then we match on `x` and `x'`, both `Empty` cases being trivial:

```
destruct x.
(* x=Empty *)
intros; exact true.
(* x = Node x1 z x2 *)
destruct x'.
(* x'=Empty *)
intros; exact false.
```

Next we proceed by case on the result of `compare z z0`:

```
(* x' = Node x'1 z0 x'2 *)
intros; case (compare z z0).
```

In each of the three cases, the recursive calls (hypothesis `H`) are handled using the `refine` tactic. We get a proof obligation expressing the decreasing of the total number of elements, which is automatically discharged by `simpl; omega` (`simpl` is required to unfold the definition of `cardinal_tree`):

```
    (* z < z0 *)
    refine (andb (H (Node x1 z Empty) x'2 _)
                 (H x2 (Node x'1 z0 x'2) _)); simpl; omega.
    (* z = z0 *)
    refine (andb (H x1 x'1 _) (H x2 x'2 _)); simpl ; omega.
    (* z > z0 *)
    refine (andb (H (Node Empty z x2) x'2 _)
                 (H x1 (Node x'1 z0 x'2) _)); simpl ; omega.
Defined.
```

Note. We could have used a single `refine` for the whole definition.

Note. It is interesting to have a look at the extracted code for a function defined using a principle such as `well_founded_induction`. We can first have a look at the extracted code for `well_founded_induction` and we recognize a fixed point operator:

```
let rec well_founded_induction x a =
  x a (fun y _ -> well_founded_induction x y)
```

When unfolding this operator and two other constants

```
Extraction NoInline andb.
Extraction Inline cardinal_rec2 Acc_iter_2 well_founded_induction_type_2.
Extraction subset.
```

we get exactly the expected ML code:

```
let rec subset x x' =
  match x with
    | Empty -> True
    | Node (x1, z0, x2) ->
        (match x' with
           | Empty -> False
           | Node (x'1, z1, x'2) ->
               (match compare z0 z1 with
                  | Lt ->
                      andb (subset (Node (x1, z0, Empty)) x'2)
                        (subset x2 (Node (x'1, z1, x'2)))
                  | Eq -> andb (subset x1 x'1) (subset x2 x'2)
                  | Gt ->
                      andb (subset (Node (Empty, z0, x2)) x'2)
                        (subset x1 (Node (x'1, z1, x'2)))))
```

Many other techniques to define functions that are not structurally recursive are described in the chapter 15 of *Interactive Theorem Proving and Program Development* [3].

1.2 The use of dependent types

Another approach to program verification in Coq consists in using the richness of the type system to express the specification of the function within its type. Actually, a type *is* a specification. In the case of ML, it is only a very poor specification (e.g. a function expects an integer and returns an integer) but in Coq one can express that a function is expecting a non-negative integer and returning a prime integer:

$$f : \{n : Z \mid n \geq 0\} \rightarrow \{p : Z \mid \texttt{prime}\ p\}$$

We are going to show how to do this in this section.

1.2.1 The subtype type `sig`

The Coq notation $\{x : A \mid P\}$ denotes the "subtype of A of values satisfying the property P" or, in a set-theoretical setting, the "subset of A of elements satisfying P". The notation $\{x : A \mid P\}$ actually stands for the application `sig` A (`fun` $x \Rightarrow P$) where `sig` is the following inductive type:

```
Inductive sig (A : Set) (P : A -> Prop) : Set :=
  exist : forall x:A, P x -> sig P
```

This inductive type is similar to the existential `ex`, apart from its sort which is `Set` instead of `Prop` (we aim at defining a function and thus its arguments and result must be informative).

In practice, we need to relate the argument and the result within a postcondition Q and thus we prefer the more general specification:

$$f : \forall\ (x : \tau_1),\ P\ x \rightarrow \{y : \tau_2 \mid Q\ x\ y\}$$

If we come back to the `min_elt` function, its specification can be the following:

```
Definition min_elt :
  forall s, ~s=Empty -> bst s ->
    { m:Z | In m s /\ forall x, In x s -> m <= x }.
```

We still have the definition issues mentioned in the previous section and thus we usually adopt a definition by proof (still with the same caution w.r.t. automatic tactics).

Note. The move of the property `bst s` from the postcondition to the precondition is not mandatory; it is only more natural.

Note. The extraction of `sig` A Q forgets the logical annotation Q and thus reduces to the extraction of type A. Said otherwise, the `sig` type can disappear at extraction time. And indeed we have

```
Coq < Extraction sig.
type 'a sig = 'a
  (* singleton inductive, whose constructor was exist *)
```

1.2.2 Variants of `sig`

We can introduce other types similar to `sig`. For instance, if we want to define a function returning two integers, such as an Euclidean division function, it seems natural to combine two instances of `sig` as we would do with two existentials `ex`:

$$\texttt{div} : \forall a\ b,\ b > 0 \rightarrow \{q \mid \{r \mid a = bq + r \wedge 0 \leq r < b\}\}$$

But the second instance of `sig` has sort `Set` and not `Prop`, which makes this statement ill-typed. Coq introduces for this purpose a variant of `sig`, `sigS` :

```
Inductive sigS (A : Set) (P : A -> Set) : Set :=
  existS : forall x:A, P x -> sig P
```

where the sole difference is the sort of `P` (`Set` instead of `Prop`). `sigS` A (`fun` $x \Rightarrow P$) is written $\{x : A\ \&\ P\}$, which allows to write

$$\texttt{div} : \forall a\ b,\ b > 0 \rightarrow \{q\ \&\ \{r \mid a = bq + r \wedge 0 \leq r < b\}\}$$

The extraction of `sigS` is naturally a pair:

```
Coq < Extraction sigS.
type ('a, 'p) sigS =
  | ExistS of 'a * 'p
```

Similarly, if we want a specification looking like

$$\{x : A \mid P\ x \wedge Q\ x\}$$

there exists and inductive "made on purpose", `sig2`, defined as

```
Inductive sig2 (A : Set) (P : A -> Prop) (Q : A -> Prop) : Set :=
  exist2 : forall x : A, P x -> Q x -> sig2 P Q
```

Its extraction is identical to the one of `sig`.

1.2.3 Specification of a boolean function: `sumbool`

A very common kind of specification is the one of a boolean function. In this case, we want to specify what are the two properties holding when the function is returning `false` and `true` respectively. Coq introduces an inductive type for this purpose, `sumbool`, defined as

```
Inductive sumbool (A : Prop) (B : Prop) : Set :=
  | left : A -> sumbool A B
  | right : B -> sumbool A B
```

It is a type similar to `bool` but each constructor contains a proof, of `A` and `B` respectively. `sumbool A B` is written `{A}+{B}`. A function checking for the empty set can be specified as follows:

$$\texttt{is\_empty} : \forall s,\ \{s = \texttt{Empty}\} + \{\neg s = \texttt{Empty}\}$$

A more general case, very common in practice, is the one of a decidable equality. Indeed, if a type `A` is equipped with an equality $\texttt{eq} : \texttt{A} \to \texttt{A} \to \texttt{Prop}$, we can specify a function deciding this equality as

$$\texttt{A\_eq\_dec} : \forall x\ y,\ \{\texttt{eq}\ x\ y\} + \{\neg(\texttt{eq}\ x\ y)\}$$

It is exactly as stating

$$\forall x\ y,\ (\texttt{eq}\ x\ y) \vee \neg(\texttt{eq}\ x\ y)$$

apart from the sort which is different. In the latter case, we have a disjunction in sort `Prop` (an excluded-middle instance for the predicate `eq`) whereas in the former case we have a "disjunction" in sort `Set`, that is a program deciding the equality.

The extraction of `sumbool` is a type isomorphic to `bool`:

```
Coq < Extraction sumbool.
type sumbool =
  | Left
  | Right
```

In practice, one can tell the Coq extraction to use ML booleans directly instead of `Left` and `Right` (which allows the ML `if-then-else` to be used in the extracted code).

Variant `sumor`

There exists a variant of `sumbool` where the sorts are not the same on both sides:

```
Inductive sumor (A : Set) (B : Prop) : Set :=
  | inleft : A -> A + {B}
  | inright : B -> A + {B}
```

This inductive type can be used to specify an ML function returning a value of type α `option`: the constructor `inright` stands for the case `None` and adds a proof of property `B`, and the constructor `inleft` stands for the case `Some` and adds a proof of property `A`. The extraction of type `sumor` is isomorphic to the ML type `option`:

```
Coq < Extraction sumor.
type 'a sumor =
  | Inleft of 'a
  | Inright
```

We can combine `sumor` and `sig` to specify the `min_elt` in the following way:

```
Definition min_elt :
  forall s, bst s ->
  { m:Z | In m s /\ forall x, In x s -> m <= x } + { s=Empty }.
```

It corresponds to the ML function turned total using an `option` type.

We can even combine `sumor` and `sumbool` to specify our ternary `compare` function:

```
Hypothesis compare : forall x y, {x<y} + {x=y} + {x>y}.
```

Note that now this single hypothesis replaces the inductive `order` and the two hypotheses `compare` and `compare_spec`.

Let us go back to the membership function on binary search trees, `mem`. We can now specify it using a dependent type:

```
Definition mem :
  forall x s, bst s -> { In x s }+{ ~(In x s) }.
```

The definition-proof starts with an induction over `s`.

```
Proof.
  induction s; intros.
  (* s = Empty *)
  right; intro h; inversion_clear h.
```

The case `s=Empty` is trivial. In the case `s=Node s1 z s2`, we need to proceed by case on the result of `compare x z`. It is now simpler than with the previous method: no more need to call for the `compare_spec` lemma, since `compare x z` contains its specification in its type.

```
  (* s = Node s1 z s2 *)
  case (compare x z); intro.
```

Similarly, each induction hypothesis (over `s1` and `s2`) is a function containing its specification in its type. We use it, when needed, by applying the `case` tactic. The remaining of the proof is easy.

Note. It is still possible to obtain the pure function as a projection of the function specified using a dependent type:

```
Definition mem_bool x s (h:bst s) := match mem x s h with
  | left _ => true
  | right _ => false
  end.
```

Then it is easy to show the correctness of this pure function (since the proof is "contained" in the type of the initial function):

```
Theorem mem_bool_correct :
  forall x s, forall (h:bst s),
  (mem_bool x s h)=true <-> In x s.
Proof.
  intros.
  unfold mem_bool; simpl; case (mem x s h); intuition.
  discriminate H.
Qed.
```

But this projection has few interests in practice.

Note. It is important to notice that each function is now given its specification when it is defined: it is no more possible to establish several properties of a same function as it was with a pure function.

1.3 Modules and functors

The adequacy of Coq as formalism to specify and prove correct purely functional ML programs extends to the module system. Indeed, Coq is equipped with a module system inspired by the module system of Objective Caml [16, 6, 7] since version 8. As Coq function types can enrich ML types with logical annotations, Coq modules can enrich ML ones.

For instance, if we want to write our finite sets library as a functor taking an arbitrary type as argument (and no more `Z` only as it was the case up to now) equipped with a total order, we start by defining a signature for this functor argument. It packs together a type `t`, an equality `eq` and an order relation `lt` over this type:

```
Module Type OrderedType.
  Parameter t : Set.
  Parameter eq : t -> t -> Prop.
  Parameter lt : t -> t -> Prop.
```

together as a decidability result for `lt` and `eq`:

```
  Parameter compare : forall x y, {lt x y}+{eq x y}+{lt y x}.
```

We also have to include some properties of `eq` (equivalence relation) and `lt` (order relation not compatible with `eq`) without which the functions over binary search trees would not be correct:

```
  Axiom eq_refl : forall x, (eq x x).
  Axiom eq_sym : forall x y, (eq x y) -> (eq y x).
  Axiom eq_trans : forall x y z, (eq x y) -> (eq y z) -> (eq x z).

  Axiom lt_trans : forall x y z, (lt x y) -> (lt y z) -> (lt x z).
  Axiom lt_not_eq : forall x y, (lt x y) -> ~(eq x y).
```

Last, we can add some `Hint` commands for the `auto` tactic to this signature, so that they will be automatically available within the functor body:

```
  Hint Immediate eq_sym.
  Hint Resolve eq_refl eq_trans lt_not_eq lt_trans.
End OrderedType.
```

Then we can write our finite sets library as a functor taking an argument `X` of type `OrderedType` as argument:

```
Module ABR (X: OrderedType).

  Inductive tree : Set :=
  | Empty
```

```
| Node : tree -> X.t -> tree -> tree.

Fixpoint mem (x:X.t) (s:tree) {struct s} : bool := ...

Inductive In (x:X.t) : tree -> Prop := ...
Hint Constructors In.

Inductive bst : tree -> Prop :=
| bst_empty : (bst Empty)
| bst_node :
   forall x (l r : tree),
   bst l -> bst r ->
   (forall y, In y l -> X.lt y x) ->
   (forall y, In y r -> X.lt x y) -> bst (Node l x r).

(* etc. *)
```

Note. The Objective Caml language provides a finite sets library based on balanced binary search trees (AVLs [2]), as a functor taking an ordered type as argument. This library implements all usual operations over sets (union, intersection, difference, cardinal, smallest element, etc.), iterators (`map`, `fold`, `iter`) and even a total order over sets allowing the construction of sets of sets by a second application of the same functor (and so on). This library has been verified using Coq by Pierre Letouzey and Jean-Christophe Filliâtre [11]. This proof exhibited a balancing bug in some functions; the code was fixed in ocaml 3.07 (and the fix verified with Coq).

Chapter 2

Verification of imperative programs: the Why tool

This chapter is an introduction to the Why tool. This tool implements a programming language designed for the verification of sequential programs. This is an intermediate language to which existing programming languages can be compiled and from which verification conditions can be computed.

Section 2.1 introduces the theory behind the Why tool (syntax, typing, semantics and weakest preconditions for its language). Then Section 2.2 illustrates the practical use of the tool on several examples and quickly describes the application of the Why tool to the verification of C and Java programs.

2.1 Underlying theory

Implementing a verification condition generator (VCG) for a realistic programming language such as C is a lot of work. Each construct requires a specific treatment and there are many of them. Though, almost all rules will end up to be instances of the five historical Hoare Logic rules [12]. Reducing the VCG to a core language thus seems a good approach. Similarly, if one has written a VCG for C and has to write another one for Java, there are clearly enough similarities to hope for this core language to be reused. Last, if one has to experiment with several logics, models and/or proof tools, this core language should ideally remain the same.

The Why tool implements such an intermediate language for VCGs, that we call HL in the following (for *Hoare Language*). Syntax, typing, semantics and weakest preconditions calculus are given below, but we first start with a tour of HL features.

Genericity. HL annotations are written in a first-order predicate syntax but are not interpreted at all. This means that HL is independent of the underlying logic in which the annotations are interpreted. The WP calculus only requires the logic to be minimal i.e. to include universal quantification, conjunction and implication.

ML syntax. HL has an ML-like syntax where there is no distinction between expressions and statements. This greatly simplifies the language—not only the syntax but also the typing and semantics. However HL has few in common with the ML family

languages (functions are not first-class values, there is no polymorphism, no type inference, etc.)

Aliases. HL is an alias-free language. This is ensured by the type checking rules. Being alias free is crucial for reasoning about programs, since the rule for assignment

$$\{P[x \leftarrow E]\}\, x \texttt{ := } E\, \{P\}$$

implicitly assumes that any variable other than x is left unmodified. Note however that the absence of alias in HL does not prevent the interpretation of programs with possible aliases: such programs can be interpreted using a more or less complex memory model made of several unaliased variables (see Section 2.2.3).

Exceptions. Beside conditional and loop, HL only has a third kind of control statement, namely exceptions. Exceptions can be thrown from any program point and caught anywhere upper in the control-flow. Arbitrary many exceptions can be declared and they may carry values. Exceptions can be used to model exceptions from the source language (e.g. Java's exceptions) but also to model all kinds of abrupt statements (e.g. C and Java's `return`, `break` or `continue`).

Typing with effects. HL has a typing with effects: each expression is given a type together with the sets of possibly accessed and possibly modified variables and the set of possibly raised exceptions. Beside its use for the alias check, this is the key to modularity: one can declare and use a function without implementing it, since its type mentions its side-effects. In particular, the WP rule for function call is absolutely trivial.

Auxiliary variables. The usual way to relate the values of variables at several program points is to used the so-called *auxiliary variables*. These are variables only appearing in annotations and implicitly universally quantified over the whole Hoare triple. Though auxiliary variables can be given a formal meaning [21] their use is cumbersome in practice: they pollute the annotations and introduce unnecessary equality reasoning on the prover side. Instead we propose the use of program *labels*—similar to those used for `goto`s—to refer to the values of variables at specific program points. This appears to be a great improvement over auxiliary variables, without loss of expressivity.

2.1.1 Syntax

Types and specifications

Program annotations are written using the following minimal first-order logic:

$$\begin{array}{rcl} t & ::= & c \mid x \mid \texttt{!}x \mid \phi(t, \ldots, t) \mid \texttt{old}(t) \mid \texttt{at}(t, L) \\ p & ::= & P(t, \ldots, t) \mid \forall x : \beta.p \mid p \Rightarrow p \mid p \wedge p \mid \ \ldots \end{array}$$

A term t can be a constant c, a variable x, the contents of a reference x (written $\texttt{!}x$) or the application of a function symbol ϕ. It is important to notice that ϕ is a function symbol belonging to the logic: it is not defined in the program. The construct $\texttt{old}(t)$ denotes

the value of term t in the precondition state (only meaningful within the corresponding postcondition) and the construct $\texttt{at}(t, L)$ denotes the value of the term t at the program point L (only meaningful within the scope of a label L).

We assume the existence of a set of *pure types* (β) in the logical world, containing at least a type `unit` with a single value `void` and a type `bool` for booleans with two values `true` and `false`.

Predicates necessarily include conjunction, implication and universal quantification as they are involved in the weakest precondition calculus. In practice, one is likely to add at least disjunction, existential quantification, negation and true and false predicates. An atomic predicate is the application of a predicate symbol P and is not interpreted. For the forthcoming WP calculus, it is also convenient to introduce an `if-then-else` predicate:

$$\begin{array}{l}\texttt{if } t \texttt{ then } p_1 \texttt{ else } p_2 \;\equiv \\ \quad (t = \texttt{true} \Rightarrow p_1) \wedge (t = \texttt{false} \Rightarrow p_2)\end{array}$$

Program types and specifications are classified as follows:

$$\begin{array}{lcl}\tau & ::= & \beta \mid \beta \;\mathsf{ref} \mid (x : \tau) \rightarrow \kappa \\ \kappa & ::= & \{p\}\,\tau\,\epsilon\,\{q\} \\ q & ::= & p; E \Rightarrow p; \ldots; E \Rightarrow p \\ \epsilon & ::= & \texttt{reads } x, \ldots, x \texttt{ writes } x, \ldots, x \texttt{ raises } E, \ldots, E\end{array}$$

A value of type τ is either an immutable variable of a pure type (β), a reference containing a value of a pure type (β ref) or a function of type $(x : \tau) \rightarrow \{p\}\,\beta\,\epsilon\,\{q\}$ mapping the formal parameter x to the specification of its body, that is a precondition p, the type τ for the returned value, an effect ϵ and a postcondition q. An effect is made of tree lists of variables: the references possibly accessed (`reads`), the references possibly modified (`writes`) and the exceptions possibly raised (`raises`). A postcondition q is made of several parts: one for the normal termination and one for each possibly raised exception (E stands for an exception name).

When a function specification $\{p\}\,\beta\,\epsilon\,\{q\}$ has no precondition and no postcondition (both being `true`) and no effect (ϵ is made of three empty lists) it can be shortened to τ. In particular, $(x_1 : \tau_1) \rightarrow \cdots \rightarrow (x_n : \tau_n) \rightarrow \kappa$ denotes the type of a function with n arguments that has no effect as long as it not applied to n arguments. Note that functions can be partially applied.

Expressions

The syntax for program expressions is given in Figure 2.1. In particular, programs contain *pure terms* (t) made of constants, variables, dereferences (written $!x$) and application of function symbols from the logic to pure terms. The syntax mostly follows ML's one. `ref` e introduces a new reference initialized with e. `loop` e {`invariant` p `variant` t} is an infinite loop of body e, invariant p and which termination is ensured by the variant t. The `raise` construct is annotated with a type τ since there is no polymorphism in HL. There are two ways to insert proof obligations in programs: `assert` $\{p\}$; e places an assertion p to be checked right before e and e $\{q\}$ places a postcondition q to be checked right after e.

$$
\begin{array}{lcl}
t & ::= & c \mid x \mid \texttt{!}x \mid \phi(t, \ldots, t) \\
e & ::= & t \\
& \mid & x \texttt{ := } e \\
& \mid & \texttt{let } x \texttt{ = } e \texttt{ in } e \\
& \mid & \texttt{let } x \texttt{ = ref } e \texttt{ in } e \\
& \mid & \texttt{if } e \texttt{ then } e \texttt{ else } e \\
& \mid & \texttt{loop } e \texttt{ \{invariant } p \texttt{ variant } t\texttt{\}} \\
& \mid & L\texttt{:}e \\
& \mid & \texttt{raise } (E\ e) : \tau \\
& \mid & \texttt{try } e \texttt{ with } E\ x \rightarrow e \texttt{ end} \\
& \mid & \texttt{assert } \{p\};\ e \\
& \mid & e\ \{q\} \\
& \mid & \texttt{fun } (x : \tau) \rightarrow \{p\}\ e \\
& \mid & \texttt{rec } x\ (x : \tau) \ldots (x : \tau) : \beta\ \texttt{\{variant } t\texttt{\}} = \{p\}\ e \\
& \mid & e\ e
\end{array}
$$

Figure 2.1: Syntax

The traditional sequence construct is only syntactic sugar for a `let-in` binder where the variable does not occur in e_2:

$$e_1\texttt{;}\ e_2 \ \equiv\ \texttt{let \_ = } e_1 \texttt{ in } e_2$$

We also simplify the `raise` construct whenever both the exception contents and the whole `raise` expression have type `unit`:

$$\texttt{raise } E \ \equiv\ \texttt{raise } (E \texttt{ void}) : \texttt{unit}$$

The traditional `while` loop is also syntactic sugar for a combination of an infinite loop and the use of an exception *Exit* to exit the loop:

$$
\begin{array}{l}
\texttt{while } e_1 \texttt{ do } e_2 \texttt{ \{invariant } p \texttt{ variant } t\texttt{\}} \ \equiv \\
\quad \texttt{try} \\
\quad\quad \texttt{loop if } e_1 \texttt{ then } e_2 \texttt{ else raise } \mathit{Exit} \\
\quad\quad \texttt{\{invariant } p \texttt{ variant } t\texttt{\}} \\
\quad \texttt{with } \mathit{Exit} \texttt{ \_ -> void end}
\end{array}
$$

Functions and programs

A program (p) is a list of declarations. A declaration (d) is either a definition introduced with `let` or a declaration introduced with `val`, or an exception declaration.

2.1.2 Typing

This section introduces typing and semantics for HL.

$$
\begin{array}{lcl}
p & ::= & \emptyset \mid d\ p \\
d & ::= & \texttt{let}\ x\ \texttt{=}\ e \\
 & \mid & \texttt{val}\ x : \tau \\
 & \mid & \texttt{exception}\ E\ \texttt{of}\ \beta
\end{array}
$$

Typing environments contain bindings from variables to types of values, exceptions declarations and labels:

$$\Gamma \ ::= \ \emptyset \mid x : \tau, \Gamma \mid \texttt{exception}\ E\ \texttt{of}\ \beta, \Gamma \mid \texttt{label}\ L, \Gamma$$

The type of a constant or a function symbol is given by the operator *Typeof*. A type τ is said to be *pure*, and we write τ `pure`, if it is not a reference type. We write $x \in \tau$ whenever the reference x appears in type τ i.e. in any annotation or effect within τ.

An effect is composed of three sets of identifiers. When there is no ambiguity we write (r, w, e) for the effect `reads` r `writes` w `raises` e. Effects compose a natural semi-lattice of bottom element $\bot = (\emptyset, \emptyset, \emptyset)$ and supremum $(r_1, w_1, e_1) \sqcup (r_2, w_2, e_2) = (r_1 \cup r_2, w_1 \cup w_2, e_1 \cup e_2)$. We also define the erasing of the identifier x in effect $\epsilon = (r, w, e)$ as $\epsilon \backslash x = (r \backslash \{x\}, w \backslash \{x\}, e \backslash \{x\})$.

We introduce the typing judgment $\Gamma \vdash e : (\tau, \epsilon)$ with the following meaning: in environment Γ the expression e has type τ and effect ϵ. Typing rules are given in Figure 2.2. They assume the definitions of the following extra judgments:

- $\Gamma \vdash \kappa$ `wf` : the specification κ is well formed in environment Γ,
- $\Gamma \vdash p$ `wf` : the precondition p is well formed in environment Γ,
- $\Gamma \vdash q$ `wf` : the postcondition q is well formed in environment Γ,
- $\Gamma \vdash t : \beta$: the logical term t has type β in environment Γ.

The purpose of this typing with effects is two-fold. First, it rejects aliases: it is not possible to bind one reference variable to another reference, neither using a `let in` construct, nor a function application. Second, it will be used when interpreting programs in Type Theory (in Section 2.1.5 below).

2.1.3 Semantics

We give a big-step operational semantics to HL. The notions of values and states are the following:

$$
\begin{array}{lcl}
v & ::= & c \mid E\ c \mid \texttt{rec}\ f\ x = e \\
s & ::= & \{(x, c), \ldots, (x, c)\}
\end{array}
$$

A value v is either a constant value (integer, boolean, etc.), an exception E carrying a value c or a closure `rec` $f\ x = e$ representing a possibly recursive function f binding x to e. For the purpose of the semantic rules, it is convenient to add the notion of closure to the set of expressions:

$$e \ ::= \ \ldots \mid \texttt{rec}\ f\ x = e$$

$$\frac{\mathit{Typeof}(c) = \beta}{\Gamma \vdash c : (\beta, \bot)} \qquad \frac{x : \tau \in \Gamma \qquad \tau \ \mathtt{pure}}{\Gamma \vdash x : (\tau, \bot)} \qquad \frac{x : \beta \ \mathsf{ref} \in \Gamma}{\Gamma \vdash !x : (\beta, \mathtt{reads}\ x)}$$

$$\frac{\Gamma \vdash t_i : (\beta_i, \epsilon_i) \quad \mathit{Typeof}(\phi) = \beta_1, \ldots, \beta_n \to \beta}{\Gamma \vdash \phi(t_1, \ldots, t_n) : (\beta, \bigsqcup_i \epsilon_i)}$$

$$\frac{x : \beta \ \mathsf{ref} \in \Gamma \qquad \Gamma \vdash e : (\beta, \epsilon)}{\Gamma \vdash x := e : (\mathtt{unit}, (\mathtt{writes}\ x) \sqcup \epsilon)}$$

$$\frac{\Gamma \vdash e_1 : (\tau_1, \epsilon_1) \qquad \tau_1 \ \mathtt{pure} \qquad \Gamma, x : \tau_1 \vdash e_2 : (\tau_2, \epsilon_2)}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : (\tau_2, \epsilon_1 \sqcup \epsilon_2)}$$

$$\frac{\Gamma \vdash e_1 : (\beta_1, \epsilon_1) \qquad \Gamma, x : \beta_1 \ \mathsf{ref} \vdash e_2 : (\tau_2, \epsilon_2) \qquad x \notin \tau_2}{\Gamma \vdash \mathtt{let}\ x = \mathtt{ref}\ e_1\ \mathtt{in}\ e_2 : (\tau_2, \epsilon_1 \sqcup \epsilon_2 \backslash x)}$$

$$\frac{\Gamma \vdash e_1 : (\mathtt{bool}, \epsilon_1) \qquad \Gamma \vdash e_2 : (\tau, \epsilon_2) \qquad \Gamma \vdash e_3 : (\tau, \epsilon_3)}{\Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : (\tau, \epsilon_1 \sqcup \epsilon_2 \sqcup \epsilon_3)}$$

$$\frac{\Gamma \vdash e : (\mathtt{unit}, \epsilon) \qquad \Gamma \vdash p\ \mathtt{wf} \qquad \Gamma \vdash t : \mathtt{int}}{\Gamma \vdash \mathtt{loop}\ e\ \{\mathtt{invariant}\ p\ \mathtt{variant}\ t\} : (\mathtt{unit}, \epsilon)}$$

$$\frac{\Gamma, \mathtt{label}\ L \vdash e : (\tau, \epsilon)}{\Gamma \vdash L{:}e : (\tau, \epsilon)}$$

$$\frac{\mathtt{exception}\ E\ \mathtt{of}\ \beta \in \Gamma \qquad \Gamma \vdash e : (\beta, \epsilon)}{\Gamma \vdash \mathtt{raise}\ (E\ e) : \tau : (\tau, (\mathtt{raises}\ E) \sqcup \epsilon))}$$

$$\frac{\mathtt{exception}\ E\ \mathtt{of}\ \beta \in \Gamma \qquad \Gamma \vdash e_1 : (\tau, \epsilon_1) \qquad \Gamma, x : \beta \vdash e_2 : (\tau, \epsilon_2)}{\Gamma \vdash \mathtt{try}\ e_1\ \mathtt{with}\ E\ x \to e_2\ \mathtt{end} : (\tau, \epsilon_1 \backslash \{\mathtt{raises}\ E\} \sqcup \epsilon_2)}$$

$$\frac{\Gamma \vdash p\ \mathtt{wf} \qquad \Gamma \vdash e : (\tau, \epsilon)}{\Gamma \vdash \mathtt{assert}\ \{p\};\ e : (\tau, \epsilon)} \qquad \frac{\Gamma \vdash e : (\tau, \epsilon) \qquad \Gamma, \mathit{result} : \tau \vdash q\ \mathtt{wf}}{\Gamma \vdash e\ \{q\} : (\tau, \epsilon)}$$

$$\frac{\Gamma, x : \tau \vdash p\ \mathtt{wf} \qquad \Gamma, x : \tau \vdash e\ \{q\} : (\tau', \epsilon)}{\Gamma \vdash \mathtt{fun}\ (x : \tau) \to \{p\}\ e\ \{q\} : ((x : \tau) \to \{p\}\, \tau'\, \epsilon\, \{q\}, \bot)}$$

$$\frac{\begin{array}{c}\Gamma' \equiv \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \qquad \Gamma' \vdash p\ \mathtt{wf} \qquad \Gamma' \vdash t : \mathtt{int} \\ \Gamma', f : (x_1 : \tau_1) \to \cdots (x_n : \tau_n) \to \{p\}\, \tau\, \epsilon\, \{q\} \vdash e\ \{q\} : (\tau, \epsilon)\end{array}}{\begin{array}{c}\Gamma \vdash\ \mathtt{rec}\ f\ (x_1 : \tau_1) \ldots (x_n : \tau_n) : \tau\ \{\mathtt{variant}\ t\} = \{p\}\ e\ \{q\} \\ : ((x_1 : \tau_1) \to \cdots (x_n : \tau_n) \to \{p\}\, \tau\, \epsilon\, \{q\}, \bot)\end{array}}$$

$$\frac{\Gamma \vdash e_1 : ((x : \tau_2) \to \{p\}\, \tau_2\, \epsilon\, \{q\}, \epsilon_1) \qquad \Gamma \vdash e_2 : (\tau_2, \epsilon_2)) \qquad \tau_2 \ \mathtt{pure}}{\Gamma \vdash e_1\ e_2 : (\tau, \epsilon_1 \sqcup \epsilon_2 \sqcup \epsilon)}$$

$$\frac{\Gamma \vdash e_1 : ((x : \beta \ \mathsf{ref}) \to \{p\}\, \tau_2\, \epsilon\, \{q\}, \epsilon_1) \qquad x_2 : \beta \ \mathsf{ref} \in \Gamma \qquad x_2 \notin \tau_2}{\Gamma \vdash e_1\ x_2 : (\tau[x \leftarrow x_2], \epsilon_1 \sqcup \epsilon[x \leftarrow x_2])}$$

Figure 2.2: Typing

In order to factor out all semantic rules dealing with uncaught exceptions, we introduce the following set of contexts R:

$$\begin{array}{rcl} R & ::= & [] \mid x \texttt{ := } R \mid \texttt{let } x \texttt{ = } R \texttt{ in } e \mid \texttt{let } x \texttt{ = ref } R \texttt{ in } e \\ & \mid & \texttt{if } R \texttt{ then } e \texttt{ else } e \mid \texttt{loop } R \ \{\texttt{invariant } p \texttt{ variant } t\} \\ & \mid & \texttt{raise } (E\ R) : \tau \mid R\ e \end{array}$$

The semantics rules are given Figure 2.3.

2.1.4 Weakest preconditions

Programs correctness is defined using a calculus of weakest preconditions. We note $wp(e, q; r)$ the weakest precondition for a program expression e and a postcondition $q; r$ where q is the property to hold when terminating normally and $r = E_1 \Rightarrow q_1; \ldots; E_n \Rightarrow q_n$ is the set of properties to hold for each possibly uncaught exception. Expressing the correctness of a program e is simply a matter of computing $wp(e, \mathsf{True})$.

The rules for the basic constructs are the following:

$$\begin{array}{rl} wp(t, q; r) & = q[\mathit{result} \leftarrow t] \\ wp(x \texttt{ := } e, q; r) & = wp(e, q[\mathit{result} \leftarrow \texttt{void}; x \leftarrow \mathit{result}]; r) \\ wp(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2, q; r) & = wp(e_1, wp(e_2, q; r)[x \leftarrow \mathit{result}]; r) \\ wp(\texttt{let } x \texttt{ = ref } e_1 \texttt{ in } e_2, q; r) & = wp(e_1, wp(e_2, q)r[x \leftarrow \mathit{result}]; r) \\ wp(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, q; r) & = wp(e_1, \texttt{if } \mathit{result} \texttt{ then } wp(e_2, q; r) \texttt{ else } wp(e_3, q; r); r) \\ wp(L\texttt{:}e, q; r) & = wp(e, q; r)[\texttt{at}(x, L) \leftarrow x] \end{array}$$

On the traditional constructs of Hoare logic, these rules simplify to the well known identities. For instance, the case of the assignment of a side-effect free expression gives

$$wp(x \texttt{ := } t, q) = q[x \leftarrow t]$$

and the case of a (exception free) sequence gives

$$wp(e_1\texttt{; } e_2, q) = wp(e_1, wp(e_2, q))$$

The cases of exceptions and annotations are also straightforward:

$$\begin{array}{rl} wp(\texttt{raise } (E\ e) : \tau, q; r) & = wp(e, r(E); r) \\ wp(\texttt{try } e_1 \texttt{ with } E\ v \rightarrow e_2 \texttt{ end}, q; r) & = wp(e_1, q; wp(e_2, q; r)[v \leftarrow \mathit{result}]) \\ wp(\texttt{assert } \{p\}\texttt{; } e, q; r) & = p \wedge wp(e, q; r) \\ wp(e\ \{q', r'\}, q; r) & = wp(e, q' \wedge q; r' \wedge r) \end{array}$$

The case of an infinite loop is more subtle:

$$wp(\texttt{loop } e\ \{\texttt{invariant } p \texttt{ variant } t\}, q; r) \quad = p \ \wedge\ \forall \omega.\ p \Rightarrow wp(L\texttt{:}e, p \wedge t < \texttt{at}(t, L); r)$$

where ω stands for the set of references possibly modified by the loop body (the `writes` part of e's effect). Here the weakest precondition expresses that the invariant must hold initially and that for each turn in the loop (represented by ω), either p is preserved by e and e decreases the value of t (to ensure termination), or e raises an exception and thus must establish r directly.

$$\frac{}{s, c \longrightarrow s, c} \qquad \frac{}{s, \texttt{!}x \longrightarrow s, s(x)} \qquad \frac{s, t_i \longrightarrow s, c_i}{s, \phi(t_1, \ldots, t_n) \longrightarrow s, \phi(c_1, \ldots, c_n)}$$

$$\frac{s, e \longrightarrow s', E\ c}{s, R[e] \longrightarrow s', E\ c} \qquad \frac{s, e \longrightarrow s', c}{s, x\ \texttt{:=}\ e \longrightarrow s' \oplus \{x \mapsto c\}, \texttt{void}}$$

$$\frac{s, e_1 \longrightarrow s_1, v_1 \quad v_1 \text{ not exc.} \quad s_1, e_2[x \leftarrow v_1] \longrightarrow s_2, v_2}{s, \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \longrightarrow s_2, v_2}$$

$$\frac{s, e_1 \longrightarrow s_1, c_1 \quad s_1 \oplus \{x \mapsto c_1\}, e_2 \longrightarrow s_2, v_2}{s, \texttt{let}\ x\ \texttt{=}\ \texttt{ref}\ e_1\ \texttt{in}\ e_2 \longrightarrow s_2, v_2}$$

$$\frac{s, e_1 \longrightarrow s_1, \texttt{true} \quad s_1, e_2 \longrightarrow s_2, v_2}{s, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \longrightarrow s_2, v_2} \qquad \frac{s, e_1 \longrightarrow s_1, \texttt{false} \quad s_1, e_3 \longrightarrow s_3, v_3}{s, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \longrightarrow s_3, v_3}$$

$$\frac{s, e \longrightarrow s', \texttt{void} \quad s', \texttt{loop}\ e\ \{\texttt{invariant}\ p\ \texttt{variant}\ t\} \longrightarrow s'', v}{s, \texttt{loop}\ e\ \{\texttt{invariant}\ p\ \texttt{variant}\ t\} \longrightarrow s'', v}$$

$$\frac{s, e \longrightarrow s', v}{s, L\texttt{:}e \longrightarrow s', v} \qquad \frac{s, e \longrightarrow s', c}{s, \texttt{raise}\ (E\ e) : \tau \longrightarrow s', E\ c}$$

$$\frac{s, e_1 \longrightarrow s_1, E'\ c \quad E' \neq E}{s, \texttt{try}\ e_1\ \texttt{with}\ E\ x \rightarrow e_2\ \texttt{end} \longrightarrow s_1, E'\ c}$$

$$\frac{s, e_1 \longrightarrow s_1, E\ c \quad s_1, e_2[x \leftarrow c] \longrightarrow s_2, v_2}{s, \texttt{try}\ e_1\ \texttt{with}\ E\ x \rightarrow e_2\ \texttt{end} \longrightarrow s_2, v_2} \qquad \frac{s, e_1 \longrightarrow s_1, v_1 \quad v_1 \text{ not exc.}}{s, \texttt{try}\ e_1\ \texttt{with}\ E\ x \rightarrow e_2\ \texttt{end} \longrightarrow s_1, v_1}$$

$$\frac{s, e \longrightarrow s', v}{s, \{p\}\ e \longrightarrow s', v} \qquad \frac{s, e \longrightarrow s', v}{s, e\ \{q\} \longrightarrow s', v}$$

$$\frac{}{s, \texttt{fun}\ (x : \tau) \rightarrow \{p\}\ e \longrightarrow s, \texttt{rec}\ \_\ x = e}$$

$$\frac{}{\begin{array}{c} s, \texttt{rec}\ f\ (x_1 : \tau_1) \ldots (x_n : \tau_n) : \tau\ \{\texttt{variant}\ t\} = \{p\}\ e \longrightarrow \\ s, \texttt{rec}\ f\ x_1 = \texttt{rec}\ \_\ x2 = \ldots \texttt{rec}\ \_\ x_n = e \end{array}}$$

$$\frac{s, e_1 \longrightarrow s_1, \texttt{rec}\ f\ x = e \quad s_1, e_2 \longrightarrow s_2, v_2 \quad s_2, e[f \leftarrow \texttt{rec}\ f\ x = e, x \leftarrow v_2] \longrightarrow s_3, v}{e_1\ e_2 \longrightarrow s_3, v}$$

Figure 2.3: Semantics

By combining this rule and the rule for the conditional, we can retrieve the rule for the usual while loop:

$$\begin{array}{ll} & wp(\texttt{while}\ e_1\ \texttt{do}\ e_2\ \{\texttt{invariant}\ p\ \texttt{variant}\ t\}, q; r) \\ = & p\ \wedge\ \forall\omega.\ p \Rightarrow \\ & wp(L\texttt{:if}\ e_1\ \texttt{then}\ e_2\ \texttt{else raise}\ E, p \wedge t < \texttt{at}(t, L), E \Rightarrow q; r) \\ = & p\ \wedge\ \forall\omega.\ p \Rightarrow \\ & wp(e_1, \texttt{if}\ \mathit{result}\ \texttt{then}\ wp(e_2, p \wedge t < \texttt{at}(t, L))\ \texttt{else}\ q, r)[\texttt{at}(x, L) \leftarrow x] \end{array}$$

Finally, we give the rules for functions and function calls. Since a function cannot be mentioned within the postcondition, the weakest preconditions for function constructs `fun` and `rec` are only expressing the correctness of the function body:

$$wp(\texttt{fun}\ (x:\tau) \rightarrow \{p\}\ e, q; r) \quad = q\ \wedge\ \forall x.\forall\rho.p \Rightarrow wp(e, \mathsf{True})$$

$$\begin{array}{l} wp(\texttt{rec}\ f\ (x_1:\tau_1)\ldots(x_n:\tau_n):\tau\ \{\texttt{variant}\ t\} = \{p\}\ e, q; r) \\ = q\ \wedge\ \forall x_1.\ldots\forall x_n.\forall\rho.p \Rightarrow wp(L\texttt{:}e, \mathsf{True}) \end{array}$$

where ρ stands for the set of references possibly accessed by the loop body (the `reads` part of e's effect). In the case of a recursive function, $wp(L\texttt{:}e, \mathsf{True})$ must be computed within an environment where f is assumed to have type $(x_1:\tau_1) \rightarrow \cdots \rightarrow (x_n:\tau_n) \rightarrow \{p \wedge t < \texttt{at}(t, L)\}\ \tau\ \epsilon\ \{q\}$ i.e. where the decreasing of the variant t has been added to the precondition of f.

The case of a function call $e1\ e2$ can be simplified to the case of an application $x_1\ x_2$ of one variable to another, using the following transformation if needed:

$$e_1\ e_2 \equiv \texttt{let}\ x_1 = e_1\ \texttt{in let}\ x_2 = e_2\ \texttt{in}\ x_1\ x_2$$

Then assuming that x_1 has type $(x:\tau) \rightarrow \{p'\}\ \tau'\ \epsilon\ \{q'\}$, we define

$$wp(x_1\ x_2, q) = p'[x \leftarrow x_2]\ \wedge\ \forall\omega.\forall\mathit{result}.(q'[x \leftarrow x_2] \Rightarrow q)[\texttt{old}(t) \leftarrow t]$$

that is (1) the precondition of the function must hold and (2) its postcondition must imply the expected property q whatever the values of the modified references and of the result are. Note that q and q' may contain exceptional parts and thus the implication is an abuse for the conjunction of all implications for each postcondition part.

2.1.5 Interpretation in Type Theory

Expressing program correctness using weakest preconditions is error-prone. Another approach consists in interpreting programs in Type Theory [9, 10] in such a way that if the interpretation can be typed then the initial imperative program is correct. It can be shown that the resulting set of proof obligations is equivalent to the weakest precondition.

The purpose of these notes is not to detail this methodology, only to introduce the language implemented in the Why tool.

2.2 The WHY tool in practice

The Why tool implements the programming language presented in the previous section. It takes annotated programs as input and generates proof obligations for a wide set of proof assistants (Coq, PVS, Isabelle/HOL, HOL 4, HOL Light, Mizar) and decision procedures (Simplify, haRVey, CVC Lite). The Why can be seen from two angles:

1. as a tool to verify *algorithms* rather than *programs*, since it implements a rather abstract and idealistic programming language. Several non-trivial algorithms have already been verified using the Why tool, such as the Knuth-Morris-Pratt string searching algorithm for instance.

2. as a tool to compute weakest preconditions, to be used as an intermediate step in the verification of existing programming languages. It has already been successfully applied to the verification of C and Java programs (as briefly sketched in the next section 2.2.3).

To remain independent of the back-end prover that will be used (it may even be several of them), the Why tool makes no assumption regarding the logic used. It uses a *syntax* of first-order predicates for annotations with no particular interpretation (apart from the usual connectives). Function symbols and predicates can be declared in order to be used in annotations, but they will be given meaning on the prover side.

2.2.1 A trivial example

Here is a small example of Why input code:

```
logic min: int, int -> int
parameter r: int ref
let f (n:int) = {} r := min !r n { r <= r@ }
```

This code declares a function symbol `min` and gives its arity. Whatever the status of this function is on the prover side (primitive, user-defined, axiomatized, etc.), it simply needs to be declared in order to be used in the following of the code. The next line declares a parameter, that is a value that is not defined but simply *assumed* to exist i.e. to belong to the environment. Here the parameter has name `r` and is an integer reference (Why's concrete syntax is very close to Ocaml's syntax). The third line defines a function `f` taking a integer `n` as argument (the type has to be given since there is no type inference in Why) and assigning to `r` the value of `min !r n`. The function `f` has no precondition and a postcondition expressing that the final value of `r` is smaller than its initial value. The current value of a reference x is directly denoted by x within annotations (not `!`x) and within postconditions x@ is the notation for `old`(x).

Let us assume the three lines code above to be in file `test.why`. Then we can produce the proof obligations for this program, to be verified with Coq, using the following command line:

```
why --coq test.why
```

A Coq file `test_why.v` is produced which contains the statement of a single proof obligation, which looks like

```
Lemma f_po_1 :
  forall (n: Z),
  forall (r: Z),
  forall (result: Z),
  forall (Post2: result = (min r n)),
  result <= r.
Proof.
(* FILL PROOF HERE *)
Save.
```

The proof itself has to be filled in by the user. If the Why input code is modified and Why run again, only the statement of the proof obligation will be updated and the remaining of the file (including the proof) will be left unmodified. Assuming that `min` is adequately defined in Coq, the proof above is trivial.

Trying an automatic decision procedure instead of Coq is as easy as running Why with a different command line option. For instance, to use Simplify [1], we type in

```
why --simplify test.why
```

A Simplify input file `test_why.sx` is produced. But Simplify is not able to discharge the proof obligation, since the meaning of `min` is unknown for Simplify:

```
Simplify test_why.sx
...
1: Invalid
```

The user can edit the header of `test_why.sx` to insert an axiom for `min`. Alternatively, this axiom can be inserted directly in the Why input code:

```
logic min: int, int -> int
axiom min_ax: forall x,y:int. min(x,y) <= x
parameter r: int ref
let f (n:int) = {} r := min !r n { r <= r@ }
```

This way this axiom will be replicated in any prover selected by the user. When using Coq, it is even possible to prove this axiom, though it is not mandatory. With the addition of this axiom, Simplify is now able to discharge the proof obligation:

```
why --simplify test.why
Simplify test_why.sx
1: Valid.
```

2.2.2 A less trivial example: Dijkstra's Dutch flag

Dijkstra's Dutch flag is a classical algorithm which sorts an array where elements can have only three different values. Assuming that these values are the three colors blue, white and red, the algorithm restores the Dutch (or French :-) national flag within the array.

This algorithm can be coded with a few lines of C, as follows:

```
typedef enum { BLUE, WHITE, RED } color;

void swap(int t[], int i, int j) { color c = t[i]; t[i] = t[j]; t[j] = c;}

void flag(int t[], int n) {
  int b = 0, i = 0, r = n;
  while (i < r) {
    switch (t[i]) {
    case BLUE: swap(t, b++, i++); break;
    case WHITE: i++; break;
    case RED: swap(t, --r, i); break;
    }
  }
}
```

We are going to show how to verify this algorithm—the *algorithm*, not the C code—using Why. First we introduce an abstract type `color` for the colors together with three values `blue`, `white` and `red`:

```
type color

logic blue : color
logic white : color
logic red : color
```

Such a new type is necessarily an *immutable* datatype. The only mutable values in Why are references (and they only contain immutable values).

Then we introduce another type `color_array` for arrays:

```
type color_array

logic acc : color_array, int -> color
logic upd : color_array, int, color -> color_array
```

Again, this is an immutable type, so it comes with a purely applicative signature (`upd` is returning a *new* array). To get the usual theory of applicative arrays, we can add the necessary axioms:

```
axiom acc_upd_eq :
  forall t:color_array. forall i:int. forall c:color.
    acc(upd(t,i,c),i) = c
axiom acc_upd_neq :
  forall t:color_array. forall i:int. forall j:int. forall c:color.
    j<>i -> acc(upd(t,i,c),j) = acc(t,j)
```

The program arrays will be references containing values of type `color_array`. In order to constraint accesses and updates to be performed within arrays bounds, we add a notion of array length and two "programs" `get` and `set` with adequate preconditions:

```
logic length : color_array -> int

axiom length_upd : forall t:color_array. forall i:int. forall c:color.
  length(upd(t,i,v)) = length(t)

parameter get :
  t:color_array ref -> i:int ->
    { 0<=i<length(t) } color reads t { result=acc(t,i) }

parameter set :
  t:color_array ref -> i:int -> c:color ->
    { 0<=i<length(t) } unit writes t { t=upd(t@,i,c) }
```

These two programs need not being defined (they are only here to insert assertions automatically), so we declare them as parameters[1].

We are now in position to define the `swap` function:

```
let swap (t:color_array ref) (i:int) (j:int) =
  { 0 <= i < length(t) and 0 <= j < length(t) }
  let c = get t i in
  set t i (get t j);
  set t j c
  { t = upd(upd(t@,i,acc(t@,j)), j, acc(t@,i)) }
```

The precondition for `swap` states that the two indices `i` and `j` must point within the array `t` and the postcondition is simply a rephrasing of the code on the model level i.e. on purely applicative arrays. Verifying the `swap` function is immediate.

Next we need to give the main function a specification. First, we need to express that the array only contains one of the three values `blue`, `white` and `red`. Indeed, nothing prevents the type `color` to be inhabitated with other values (there is no notion of inductive type in Why logic, since it is intended to be a common fragment of many tools, including many with no primitive notion of inductive types). So we define the following predicate `is_color`:

```
predicate is_color(c:color) = c=blue or c=white or c=red
```

Note that this predicate is given a *definition* in Why.

Second, we need to express the main function postcondition that is, for the final contents of the array, the property of being "sorted" but also the property of being a permutation of the initial contents of the array (a property usually neglected but clearly as important as the former). For this purpose, we introduce a predicate `monochrome` expressing that a set of successive elements is monochrome:

```
predicate monochrome(t:color_array, i:int, j:int, c:color) =
  forall k:int. i<=k<j -> acc(t,k)=c
```

[1]The Why tool actually provides a datatype of arrays, exactly in the way we are doing it here, and even a nice syntax for array operations.

For the permutation property, we only *declare* a predicate that will be defined on the prover side, whatever the prover is:

```
logic permutation : color_array, color_array, int, int -> prop
```

To be able to write down the code, we still need to translate the `switch` statement into successive tests, and for this purpose we need to be able to decide equality of the type `color`. We can declare this ability with the following parameter:

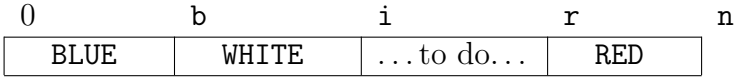
```
parameter eq_color  :
  c1:color -> c2:color -> {} bool { if result then c1=c2 else c1<>c2 }
```

Note that the meaning of = within annotations has nothing to do with a boolean function deciding equality that we could use in our programs.

We can now write the Why code for the main function:

```
let dutch_flag (t:color_array ref) (n:int) =
  { length(t) = n and forall k:int. 0 <= k < n -> is_color(acc(t,k)) }
  let b = ref 0 in
  let i = ref 0 in
  let r = ref n in
  while !i < !r do
     if (eq_color (get t !i) blue) then begin
       swap t !b !i;
       b := !b + 1;
       i := !i + 1
     end else if (eq_color (get t !i) white) then
       i := !i + 1
     else begin
       r := !r - 1;
       swap t !r !i
     end
  done
  { (exists b:int. exists r:int.
       monochrome(t,0,b,blue) and
       monochrome(t,b,r,white) and
       monochrome(t,r,n,red))
    and permutation(t,t@,0,n-1) }
```

As given above, the code cannot be proved correct, since a loop invariant is missing, and so is a termination argument. The loop invariant must maintain the current situation, which can be depicted as



But the loop invariant must also maintain less obvious properties such as the invariance of the array length (which is obvious since we only performs `upd` operations over the array, but we need not to loose this property) and the permutation w.r.t. the initial array. The termination is trivially ensured since `r-i` decreases at each loop step and is bound by 0. Finally, the loop is annotated as follows:

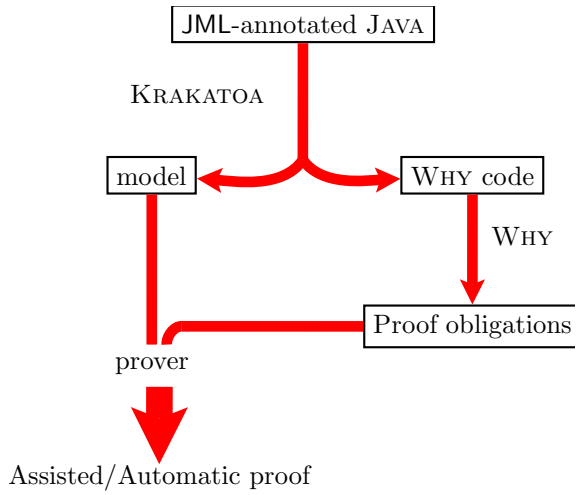


Figure 2.4: Verifying Java programs using Krakatoa and Why

```
...
while !i < !r do
   { invariant 0 <= b <= i and i <= r <= n and
               monochrome(t,0,b,blue) and
               monochrome(t,b,i,white) and
               monochrome(t,r,n,red) and
               length(t) = n and
               permutation(t,t@init,0,n-1)
     variant r - i }
...
```

We can now proceed to the verification of the program, which causes no difficulty (most proof obligations are even discharged automatically by Simplify).

2.2.3 Application to the verification of C and Java programs

The Why tool is applied to the verification of C and Java programs, as the back-end of two open-source tools CADUCEUS [14] and KRAKATOA [8] respectively. Both tools are based on the same kind of model, following Bornat [4], and handle almost all ANSI C and all sequential Java respectively. As far as KRAKATOA is concerned, Java programs are annotated using the Java Modeling Language (JML) [15] and thus KRAKATOA is very similar to tools like LOOP [22] or JACK [5]. An overview of the KRAKATOA-Why combination is given Figure 2.4. The combination with CADUCEUS is very similar.

Bibliography

[1] The Simplify decision procedure (part of ESC/Java). `http://research.compaq.com/SRC/esc/simplify/`.

[2] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics–Doklady*, 3(5):1259–1263, September 1962.

[3] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Texts in Theoretical Computer Science. An EATCS Series. Springer Verlag, 2004. `http://www.labri.fr/Perso/~casteran/CoqArt/index.html`.

[4] Richard Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.

[5] Lilian Burdy and Antoine Requet. Jack: Java Applet Correctness Kit. In *Gemplus Developers Conference GDC'2002*, 2002. See also `http://www.gemplus.com/smart/r_d/trends/jack.html`.

[6] Jacek Chrzaszcz. Implementing modules in the system Coq. In *16th International Conference on Theorem Proving in Higher Order Logics*, University of Rome III, September 2003.

[7] Jacek Chrzaszcz. *Modules in Type Theory with generative definitions*. PhD thesis, Warsaw University and Université Paris-Sud, 2003. To be defended.

[8] Claude Marché, Christine Paulin and Xavier Urbain. The Krakatoa Tool for JML/Java Program Certification. Submitted to JLAP. `http://www.lri.fr/~marche/krakatoa/`.

[9] J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types*. Thèse de doctorat, Université Paris-Sud, July 1999.

[10] J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4):709–745, July 2003. English translation of [9].

[11] Jean-Christophe Filliâtre and Pierre Letouzey. Functors for Proofs and Programs. In *Proceedings of The European Symposium on Programming*, Barcelona, Spain, March 29-April 2 2004. Voir aussi `http://www.lri.fr/~filliatr/fsets/`.

[12] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, 1969. Also in [13] pages 45–58.

[13] C. A. R. Hoare and C. B. Jones. *Essays in Computing Science*. Prentice Hall, 1989.

[14] Jean-Christophe Filliâtre and Claude Marché. The Caduceus tool for the verification of C programs. `http://why.lri.fr/caduceus/`.

[15] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, Iowa State University, 2000.

[16] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[17] Pierre Letouzey. A New Extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002*, volume 2646 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[18] Pierre Letouzey. *Programmation fonctionnelle certifiée en Coq*. PhD thesis, Université Paris Sud, 2003. To be defended.

[19] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In Association for Computing Machinery, editor, *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, January 1989.

[20] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. PhD thesis, Université Paris 7, January 1989.

[21] T. Schreiber. Auxiliary Variables and Recursive Procedures. In *TAPSOFT'97: Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 697–711. Springer-Verlag, April 1997.

[22] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi (eds.), editors, *Tools and Algorithms for the Construction and Analysis of Software (TACAS*, volume 2031 of *LNCS*, pages 299–312. Springer-Verlag, 2001.

PART V:

Dependently Typed Programming

PART VI:

Formalisation of Mathematics

Herman Geuvers:

Fundamental theory of algebra in Coq

Types Summer School
Gothenburg Sweden August 2005

Lecture 5: FTA, the Fundamental Theorem of ALgebra
C-CoRN, The Constructive Coq Repository @ Nijmegen

Herman Geuvers, Luis Cruz-Filipe, Freek Wiedijk, Milad Niqui,
Jan Zwanenburg,
Randy Pollack, Iris Loeb, Bas Spitters, Sebastien Hinderer,
Henk Barendregt, Dan Synek
Radboud University Nijmegen, NL

What, Where, Why

- What: A coherent library of formalized mathematics
- Where: @ Nijmegen (NL), but possibly users and contributors from all over the world.
- Why: formalize mathematics in a uniform way.

What? Content

- Algebraic Hierarchy: monoids, rings, (ordered) fields, ...
- Tactics, esp. for equational reasoning
- Real number structures: axiomatically as complete Archimedean ordered fields.
- Model of $\mathbb{R}$ + proof that two real number structures are isomorphic + alternative axioms
- Generic results about $\mathbb{R}$ and $\mathbb{R}$-valued functions
- (Original) FTA-library: definition of $\mathbb{C}$ and proof of FTA
- Real analysis following Bishop: Continuity, differentiability and integrability, Rolle's Theorem, Taylor's Theorem, FTC. The exponential and trigonometric functions, logarithms and inverse trigonometric functions.

The sizes of the C-CoRN library:

Description	Size (Kb)	% of total
Algebraic Hierarchy (incl. tactics)	533	26.4
Real Numbers (incl. Models)	470	23.3
FTA (incl. Complex Numbers)	175	8.7
Real Analysis (incl. Transc. Fns.)	842	41.6
Total	2020	100

Why? Aims

- Not one (isolated) big fancy theorem, but create a library: "Mexican hat"

Sets and Basics	41 kb
Algebra (upto Ordered Fields)	165 kb
Reals	52 kb
Polynomials	113 kb
Real-valued functions / Basic Analysis	30 kb
Complex numbers	98 kb
FTA proof	70 kb
Construction of $\mathbb{R}$ (Niqui)	309 kb
Rational Tactic	49 kb

Aims ctd.

- Make interaction between different fields of mathematics possible.
- Reusable by others: take care of documentation, presentation, notation, searching
- Constructive(?)
  Finer analysis of mathematics, esp. analysis: reals are (potentially) infinite objects; computational content.
- Formalizing math. on a computer is fun, but also has benefits:
  - Correctness guaranteed.
  - Exchange of 'meaningful' mathematics.
  - Finding mathematical results.

Aims ctd.

- Investigate the current limitations.
- Try to manage this project. Three sequential/parallel phases:

Mathematical proof	LaTeX document (lots of details)
Theory development	Coq file (just defs and statements of lemmas)
Proof development	Coq file (proofs filled in)

Try to keep these phases consistent!

Problems ?

- Idiosyncrasies of 'the' Proof Assistant.
- Verbosity of formalized mathematics.
- Access to the formalized mathematics.

Methodology

Work in a systematic way (CVS):

- Documentation: what has been formalized; notations; definitions; tactics.
- Structuring: Group Lemmas and Def's according to mathematical content; Name Lemmas and Def's consistently.
- Axiomatic Approach: C-CoRN aims at generality.
- Automation: Develop tactics for specific fields of mathematics

A brief look into C-CoRN

- (Constructive) Setoids
- Algebraic Hierarchy
- Partial Functions
- $\mathbb{R}$
- FTA proof
- Automation via Reflection

Setoids

How to represent the notion of set?
Note: A set is not just a type, because
$M : A$ is decidable whereas $t \in X$ is undecidable
A setoid is a pair $[A, =]$ with

- A : Set,
- $= : A \to (A \to \text{Prop})$ an equivalence relation over A

A setoid function is an $f{:}A \to B$ such that

$$\forall x, y{:}A.(x =_A y) \to (f\ x) =_B (g\ y).$$

Here: Constructive Setoids
Apartness $\#$ as basic:

$$\begin{gathered} x = y \leftrightarrow \neg(x \# y) \\ x \# y \to (x \# z) \vee (y \# z) \\ \neg(x \# x) \\ x \# y \to y \# x \end{gathered}$$

A constructive setoid function is an $f{:}A \to B$ such that

$$\forall x, y{:}A.(f\ x) \#_B (g\ y) \to (x \#_A y).$$

Strong extensionality

The algebraic hierarchy

- We deal with real numbers, complex numbers, polynomials, ...
- Many of the properties we use are generic and algebraic.
- To be able to reuse results and notation we have defined a hierarchy of algebraic structures.
- Basic level: constructive setoids.
- Next level: semi-groups, $\langle S, + \rangle$, with S a setoid and $+$ an associative binary operation on S.

Structures and Coercions

```
Record CMonoid : Type :=
  { m_crr   :> CSemi_grp;
    m_proof :  (Commutative m_crr (sg_op m_crr))
       /\ (IsUnit m_crr (sg_unit m_crr) (sg_op m_crr))
  }.
```

- A monoid is now a tuple $\langle\langle\langle S, =_S, r\rangle, a, f, p\rangle, q\rangle$
  If `M : Monoid`, the carrier of `M` is `(crr(sg_crr(m_crr M)))`
  Nasty !!
  $\Rightarrow$ We want to use the structure `M` as synonym for the carrier set `(crr(sg_crr(m_crr M)))`.
  $\Rightarrow$ The maps `crr`, `sg_crr`, `m_crr` should be left implicit.
- The notation `m_crr :> Semi_grp` declares the coercion `m_crr : Monoid >-> Semi_grp`.

Inheritance via Coercions

We have the following coercions.

```
OrdField >-> Field >-> Ring >-> Group
            Group >-> Monoid >-> Semi_grp >-> Setoid
```

- All properties of groups are inherited by rings, fields, etc.
- Also notation is inherited:

  `x[+]y`

  denotes the addition of `x` and `y` for `x,y:G` from any semi-group (or monoid, group, ring,...) `G`.
- The coercions must form a tree, so there is no real multiple inheritance:
  E.g. it is not possible to define rings in such a way that it inherits both from its additive group and its multiplicative monoid.

Partiality: Proof terms inside objects

- The 'subtype' $\{t : A \mid (P\ t)\}$ is defined as the type of pairs $\langle t, p\rangle$ where $t : A$ and $p : (P\ t)$.
  Notation: $\Sigma x{:}A.P\ x$
- A partial function is a function on a subtype
  E.g. $(-)^{-1} : \Sigma x{:}\mathbb{R}.x \neq 0 \to \mathbb{R}$.
  If $x : \mathbb{R}$ and $p : x \neq 0$, then $\frac{1}{\langle x,p\rangle} : \mathbb{R}$.
- A partialfunction must be proof-irrelevant, i.e.
  if $p : t \neq 0$ and $q : t \neq 0$, then $\frac{1}{\langle t,p\rangle} = \frac{1}{\langle t,q\rangle}$.
- For practical (Coq) purposes we "Curry" partial functions and take
  $(-)^{-1} : \Pi x{:}\mathbb{R}.(x \neq 0) \to \mathbb{R}$.

The Real Numbers in Coq:

- Axiomatic: a 'Real Number Structure' is a
  Cauchy-complete Archimedean ordered field.
- Prove FTA 'for all real numbers structures'.
- Construct a model to show that real number structures exist. (Cauchy sequences over an Arch. ordered field, say $\mathbb{Q}$)
- Prove that any two real number structures are isomorphic.

Axioms for Real Numbers:

- Cauchy sequences over Field F:
  g : nat $\to F$ is Cauchy if
  $$\forall \varepsilon{:}F_{>0}.\exists N{:}\mathbb{N}.\forall m \geq N(|g_m - g_N| < \varepsilon)$$
- All Cauchy sequences have a limit:
  $$\text{SeqLim} : (\Sigma g{:}\text{nat}\to F.\text{Cauchy}\, g) \to F$$
  $$\text{CauchyProp} : \forall g{:}\text{nat}\to F.(\text{Cauchy}\, g) \to \forall \varepsilon{:}F_{>0}.\exists N{:}\mathbb{N}.\forall m \geq N.(|g_m - (\text{SeqLim}\, g)| < \varepsilon)$$
- Axiom of Archimedes: (there are no non-standard elements)
  $$\forall x{:}F.\exists n{:}\mathbb{N}(n > x)$$

NB: The axiom of Archimedes proves that 'ε-Cauchy sequences' and '$\frac{1}{k}$-Cauchy sequences' coincide (similar for limits)

Consequences of the Axiomatic approach:

- We don't construct $\mathbb{R}$ out of $\mathbb{Q}$, so we don't have $\mathbb{Q} \subset \mathbb{R}$ on with $=$ decidable on $\mathbb{Q}$.
- We did not want to 'define' $\mathbb{Q} \subset \mathbb{R}$.
- Instead: modify the proof by introducing fuzziness:
  Instead of having to decide
  $$x < y \vee x = y \vee x > y,$$
  all we need to establish is whether (for given $\varepsilon > 0$)
  $$x < y + \varepsilon \vee x > y - \varepsilon$$
  which we may write as
  $$x \leq_\varepsilon y \vee x \geq_\varepsilon y$$
  This is decidable, due to the cotransitivity of the order relation:
  $$x < y \Rightarrow x < z \vee z < y$$

Intermezzo Program Extraction

The logic of Coq (and most type theories) is constructive. This implies that

if $\vdash \forall x{:}A\exists y{:}B.R\,x\,y$, then there is a term f such that $\vdash \forall x{:}A.R\,x\,(f\,x)$.

Application: From a proof term of $\forall x \in \text{nat}.\exists y \in \text{nat}.x+x \leq y$ one can extract

- a term (Coq-program) f : nat$\to$nat,
- a proof of $\forall x{:}\text{nat}.x + x \leq f\,x$ (correctness of f)

Strengthening

if $\vdash \forall x{:}A.P\,x \vee \neg P\,x$ and $\vdash \forall x{:}A.P\,x \to \exists y{:}B.R\,x\,y$, then there is a term f such that $\vdash \forall x{:}A.P\,x \to R\,x\,(f\,x)$.

Example
$$\forall l{:}\text{list}.l \neq \texttt{nil} \to \exists n{:}\text{nat}.n \leq l \wedge n \in l$$

Pros/Cons of the Axiomatic approach:
Pros:

- "Plug-in" arbitrary (your own pet) model to extract algorithm.
- Work abstractly: reuse

Cons (?):

- Choice of axioms? Don't try to be minimal! E.g.maximum function should be added.
- Can we get "good" algorithms when we work abstractly?

FTA: The classical FTA proof
Suppose $|f(z)|$ is minimal with $|f(z)| \neq 0$.
We construct a z_0 with $|f(z_0)| < |f(z)|$.
We may assume that the minimum is reached for $z = 0$.

$$f(x) = a_0 + a_k x^k + O(x^{k+1})$$

with a_k the first coefficient that's not 0.
Now take

$$z_0 := \varepsilon \sqrt[k]{-\frac{a_0}{a_k}}$$

with $\varepsilon \in \mathbb{R}_{>0}$.
If ε is small enough, the part $O\left(z_0^{k+1}\right)$ will be negligible and we get a $z_0 \neq 0$ for which

$$|f(z_0)| = a_0 + a_k \left(\varepsilon \sqrt[k]{-\frac{a_0}{a_k}}\right)^k = a_0(1 - \varepsilon^k) < |f(0)|$$

The constructive FTA proof
Define an algorithm
Given $z \in \mathbb{C}$, construct a sequence $z, z_0, z_1, \ldots$ going to the root.
Problem: in the definition

$$z_0 := \varepsilon \sqrt[k]{-\frac{a_0}{a_k}}$$

- ε must be small enough to neglect $O\left(z_0^{k+1}\right)$
- ε must be large enough to reach the root.

Solution (Kneser): write

$$f(x) = a_0 + a_k x^k + \text{other terms}$$

and find k and z_0 such that $|a_k||z_0|^k$ is big enough w.r.t. the other terms and small enough compared to $|a_0|$.

Automation via Computation
Poincaré Principle (Barendregt)

> "An equality involving a computation does not require a proof"

In type theory: if $t = q$ by evaluation (computing an algorithm), then this is a trivial equality, proved by reflexivity.
This is made precise by the conversion rule:

$$\frac{\Gamma \vdash M : A \quad A =_{\beta\iota\delta} B}{\Gamma \vdash M : B}$$

Can one actually use the programming power of Type Theory when formalizing mathematics?

Yes. For automation: replacing a proof obligation by a computation

Reflection Suppose

- We have a class of problems with a syntactic encoding as a data type, say via the type Problem.
  Example: equalities between expressions over a group
  Then the syntactic encoding is

```
Inductive E : Set :=
   evar  : nat -> E
 | eone  : E
 | eop   : E -> E -> E
 | einv  : E -> E
```

- We have a decoding function $[\![-]\!] : \text{Problem} \to \text{Prop}$
- We have a decision function $\text{Dec} : \text{Problem} \to \{0, 1\}$
- We can prove Ok : $\forall p{:}\text{Problem}((\text{Dec}(p) = 1) \to [\![p]\!])$

To verify P (from the class of problems):

- Find a p : Problem such that $[\![p]\!] = P$.
- Then $\text{Dec}(p)$ yields either 1 or 0
- If $\text{Dec}(p) = 1$, then we have a proof of P (using Ok)
- If $\text{Dec}(p) = 0$, we obtain no information about P (it 'fails')

Note: if Dec is complete:

$$\forall p{:}\text{Problem}((\text{Dec}(p) = 1) \leftrightarrow [\![p]\!])$$

then $\text{Dec}(p) = 0$ yields a proof of $\neg P$.
This can be made into a tactic, e.g. `Rational`, that proves equalities between rational expressions.

Related Work:

- Mizar largest library of formalized math., MML (Trybulec)
- HOL-light (Harrisson)
- Isabelle (Fleuriot, non-standard reals)
- Nuprl (Howe, constructive á la Bishop)
- Classical Reals in Coq (Mayero)
- Minlog (Schwichtenberg)
- FOC (Hardin, Rioboo)

Some Conclusions:

- Real mathematics, involving algebra and analysis can be formalised completely within a theorem prover (Coq).
- Setting up a basic library and some good proof automation procedures is a large part of the work.
- Library can be reused: Luis Cruz-Filipe proved FTC (and more).
- Extracting algorithms (e.g. for FTA) requires a further analysis of the proof (Luis Cruz-Filipe, Bas Spitters).
- In the end, the computational behaviour of algorithms should depend mainly on the representation of the reals.

Freek Wiedijk:

Formalisation of mathematics

Chapter 1

Writing a Mizar article in nine easy steps

After reading this chapter and doing the exercises, you will be able to write a basic Mizar article all by yourself.

The chapter presents an explanation of the nine steps needed to finish a Mizar article. For each step it also shows how this turns out for a specific example. There are exercises too, so you can test your understanding.

To be able to read this chapter you should know some basic mathematics but not much. You should know a little bit about:

- first order predicate logic
- some basic set theory

Mizar is based on set theory on top of predicate logic, that's why.

The most difficult part of writing Mizar is finding what you need in the MML which is the name of the *Mizar Mathematical Library*. Unfortunately that problem has no easy solution. You will discover that apart from that, writing a Mizar article is straight-forward.

1.1 Step 1: the mathematics

Before you start writing a Mizar article you need to decide what it should be about. It's very important not to start writing the article too soon! A proof that takes five minutes to explain – after which people look into the air and then mumble: 'hm, yes, I see' – takes about a week to formalize. So every improvement to the mathematics that you can make *before* you start formalizing will pay off very much.

The example As a running example in this chapter we write a small Mizar article called `my_mizar.miz`. It is about *Pythagorean triples*. Those are triples

of natural numbers $\{a, b, c\}$ for which holds that:

$$a^2 + b^2 = c^2$$

The Pythagorean theorem tells us that these triples correspond to square angled triangles that have sides of integer length. The best known of these triples are $\{3, 4, 5\}$ and $\{5, 12, 13\}$ but there are an infinite number of them.

The theorem that we will formalize says that all Pythagorean triples are given by the formula:

$$a = n^2 - m^2 \qquad b = 2mn \qquad c = n^2 + m^2$$

or are multiples of such a triple. For instance the triple $\{3, 4, 5\}$ is given by $n = 2$, $m = 1$, and the triple $\{5, 12, 13\}$ is given by $n = 3$, $m = 2$.

The proof of the theorem is straight-forward. You only needs to consider the case where a and b are relative prime. Some thought about parity gives that one of the a and b has to be even, and that the other and c is odd. Then if b is the even number of the two we get that:

$$\left(\frac{b}{2}\right)^2 = \frac{c^2 - a^2}{4} = \left(\frac{c-a}{2}\right)\left(\frac{c+a}{2}\right)$$

But if the product of two relative prime numbers is a square, then both of those numbers are squares. So you get:

$$\frac{c-a}{2} = m^2 \qquad \frac{c+a}{2} = n^2$$

which leads to the required formula.

Exercise 1.1.1 Study the Mizar Mathematical Library and try to decide whether Mizar is equally suited to all fields of mathematics and computer science, or that Mizar is best suited to certain subjects. In the latter case, what is most easily formalized in Mizar? And what least?

Exercise 1.1.2 We mentioned the *Pythagorean theorem* which says that the sides a, b and c of a square angled triangle satisfy $a^2 + b^2 = c^2$. Try to find this theorem in the MML. If it's there, in what article does it occur and what is the theorem's reference? If not, or if you don't like the version that you find (exercise for when you finished this chapter): write an article that proves it.

1.2 Step 2: the empty Mizar article

In order to write a Mizar article you need a working Mizar system and a file to put the Mizar article in. In fact, as you will find out, you will need *two* files: a `.miz` file for your article and a `.voc` file for its vocabulary.

In this tutorial we follow the Windows conventions of naming files. For Mizar under Unix the backslashes should go the other way. So what is called `text\my_mizar.miz` here, under Unix should be `text/my_mizar.miz`.

We will assume that you already have a properly installed Mizar system. For a description of how to install Mizar see the `readme.txt` file that is in the Mizar distribution (under Unix it is called `README`).

To write your article, you need to be in a directory that has two subdirectories called `text` and `dict`. If those directories don't exist yet, make them. Put in the `text` directory an empty file called `my_mizar.miz` and put in the `dict` directory an empty file called `my_mizar.voc`. (Replace the `my_mizar` with a more appropriate name if you are writing an article of your own.)

The smallest legal `.voc` file is empty but the smallest legal `.miz` file looks like this:

```
environ
begin
```

Use your favorite text editor to put those two lines in the `my_mizar.miz` file. Then check it using the command:

```
mizf text\my_mizar.miz
```

It will both check syntax and mathematics of your article. If everything is well this command prints something like:

```
Make Environment, Mizar Ver. 7.0.01 (Win32/FPC)
Copyright (c) 1990,2004 Association of Mizar Users
-Vocabularies-Constructors-Clusters-Notation

Verifier, Mizar Ver. 7.0.01 (Win32/FPC)
Copyright (c) 1990,2004 Association of Mizar Users
Processing: text\my_mizar.miz

Parser   [   2]   0:00
Analyzer   0:00
Checker  [   1]   0:00
Time of mizaring: 0:00
```

meaning that this 'article' contains no errors. (If you use the `emacs` editor with its Mizar mode you don't need to type the `mizf` command. In that case all you need to do to check the file is hit `C-c RET`.)

Apart from the `my_mizar.miz` file, the `text` directory now contains 25 other files. You never need to look inside them. They are only used internally by the Mizar system.

The next step is to connect the `.miz` file to the `.voc` vocabulary. Add a `vocabularies` directive to your article:

```
environ
 vocabularies MY_MIZAR;

begin
```

(Important: the name of the vocabulary has to be written in capitals! This is a remnant of Mizar's DOS origins.)

Now the article is ready to be written. For the moment add some line of text at the end of the file just to find out what happens if the article contains errors:

```
environ
 vocabularies MY_MIZAR;

begin
 hello Mizar!
```

After you run the `mizf text\my_mizar.miz` command again, Mizar will insert the error messages *inside* your file. It will put a `*` with a number below any error. Also there will be an explanation of those error numbers at the end of the file. So after running the `mizf` command again your file looks like:

```
environ
 vocabularies MY_MIZAR;

begin
 hello Mizar!
::>  *143,321

::> 143: No implicit qualification
::> 321: Predicate symbol or "is" expected
```

There is no need to try to understand those two error messages, because of course `hello Mizar!` is not legal Mizar. So remove this line, now that you have seen what Mizar thinks of it. You don't need to remove the error message lines. They vanish the next time you run `mizf`.

The lines containing error messages can be recognized because they start with `::>`. Mizar only gives you error numbers. It never prints anything about why it thinks it is wrong. Sometimes this lack of explanation is frustrating but generally Mizar errors are quite obvious.

Exercise 1.2.1 The minimal Mizar article that we showed is 2 lines long. What is the shortest article in the MML? What is the longest? What is the average number of lines in the articles in the MML?

As a rule of thumb an article less than a 1000 lines is considered too short to be submitted to the MML. However several articles in the MML are less than a 1000 lines long since articles are shortened during revision of the MML. How many of the articles in the MML are currently shorter than a 1000 lines?

Exercise 1.2.2 Copy an article from the MML to your private `text` directory. Check that the Mizar checker `mizf` processes it without error messages. Experiment with making small modifications to it and see whether Mizar can detect where you tampered with it.

Put one screen-full of non-Mizar text – for instance from a Pascal or C program – somewhere in the middle of the article. How many error messages

will you get? Can Mizar sufficiently recover from those errors to check the second half of the file reasonably well?

1.3 Step 3: the statement

To start translating your mathematics to Mizar you need to write the theorem that you want to prove in Mizar syntax.

There are two things about Mizar syntax that are important for you to note:

- There are no spelling variants in Mizar. Although Mizar resembles natural language a lot, it is a formal language and there are no possibilities to choose between phrasings. For example:

  `and` means something different from `&`

  `not` means something different from `non`

  `such that` means something different from `st`

  `assume` means something different from `suppose`

  `NAT` means something different from `Nat` means something different from `natural`

  So you should really pay attention to the exact keywords in the Mizar syntax. It's not enough if it resembles it.

  The only exception to this rule is that `be` and `being` are alternative spellings. So although it's more natural to write `let X be set` and `for X being set holds ...` you are also allowed to write `let X being set` and `for X be set holds ...`

- There is no distinction in Mizar between 'function notation' and 'operator notation'. In most programming languages something like `f(x,y)` and `x+y` are syntactically different things. In Mizar this distinction doesn't exist. In Mizar *everything* is an operator. If you write `f(x,y)` in Mizar then it really is a 'operator symbol' `f` with zero left arguments and two right arguments.

  Similarly predicate names and type names can be any string of characters that you might wish. You can mix letters, digits and any other character you might like in them. So for instance if you want to call a predicate `\/-distributive` you can do so in Mizar. And it will be one 'symbol'.

  If you are not sure what characters go together as a symbol in a Mizar formula, you can go to the web pages of the MML abstracts. In those pages the symbols are hyperlinks that point to definitions. So just click on them to find out what symbol they are.

To write Mizar you need to know how to write terms and how to write formulas.

We tackle this in top-down fashion. First we describe how to translate formulas from predicate logic into Mizar. Then we describe how the syntax of Mizar terms works.

1.3.1 Formulas

Here is a table that shows all you want to know about how to write predicate logic in Mizar:

$\bot$	`contradiction`
$\neg\phi$	`not` ϕ
$\phi \wedge \psi$	ϕ `&` ψ
$\phi \vee \psi$	ϕ `or` ψ
$\phi \Rightarrow \psi$	ϕ `implies` ψ
$\phi \Leftrightarrow \psi$	ϕ `iff` ψ
$\exists x.\psi$	`ex x st` ψ
$\forall x.\psi$	`for x holds` ψ
$\forall x.(\phi \Rightarrow \psi)$	`for x st` ϕ `holds` ψ

There is no special way to write $\top$ in Mizar. One usually writes `not contradiction` for this. Note that with the quantifiers it should be `st` and not `such that`. Also note that `for x st` ϕ `holds` ψ is just 'syntactic sugar' for `for x holds (`ϕ `implies` ψ`)`. After the parser processed it, the rest of the system will not see any difference between those two formulas.

Using this table you now can write logical formulas in Mizar.

There is one more thing that you need to know to write Mizar formulas in practice. Mizar is a *typed* language. We will discuss Mizar types in detail in Section 1.3.4 below, but the types turn up in the predicate logic formulas too. All variables that you use in formulas need to have a type. There are two ways to give a variable a type:

- Put the type in the formula using a `being` type attribution. For instance you can write an existential formula about natural numbers m and n as:

  ```
  ex m,n being Nat st ...
  ```

- Give the type for the variable with a `reserve` statement. This doesn't introduce a variable, it just introduces a notation convention. If you use a `reserve` statement you can leave out the type in the formula:

  ```
  reserve m,n for Nat;
  ex m,n st ...
  ```

  This way of typing variables in formulas is much more convenient than explicitely typing the variables and so it is the method that is generally used.

Exercise 1.3.1 Translate the following Mizar formulas into predicate logic notation:

ϕ `iff not not` ϕ

`not (`ϕ `&` ψ`) implies not` ϕ `or not` ψ

ex x st ϕ implies ψ

for x st for y st ϕ holds ψ holds χ

To do this exercise you need to know the priorities of the logical operators in Mizar. They are the usual ones in predicate logic:

ex/for < implies/iff < or < & < not

This means that you should read not ϕ implies ψ as (not ϕ) implies ψ (because the not binds stronger), but that ex x st ϕ implies ψ means ex x st (ϕ implies ψ).

Exercise 1.3.2 Translate the following predicate logic formulas into Mizar syntax:

$\neg\phi \Leftrightarrow \neg\neg\neg\phi$

$\neg(\phi \vee \psi) \Rightarrow (\neg\phi \wedge \neg\psi)$

$\exists\, x.\, (\phi \wedge \psi)$

$\exists\, x.\, \big(\big(\exists\, y.\, (\phi \Rightarrow \psi)\big) \Rightarrow \chi\big)$

1.3.2 Relations

You still need to know how to write atomic formulas. In Mizar there are two ways to write those:

- If R is a *predicate* symbol, write:

  $$x_1\,,\, x_2\,,\, \ldots,\, x_m\; R\; x_{m+1}\,,\, \ldots,\, x_{m+n}$$

  Both m or n might be 0, in which case this becomes prefix or postfix notation. Note that postfix notation can have more than one argument, as for instance in x,y are_relative_prime. Please note that brackets around the arguments of the predicate are not allowed.

- If $\mathcal{T}$ is a *type*, or the adjectives part of a type, write:

  $$x \text{ is } \mathcal{T}$$

  For instance you can write x is prime. We will discuss types in Section 1.3.4 below.

To find out what relations are available to you, you will need to browse the MML. Here is a short list of some of the most frequent used relations to get you started.

$=$	`=`
$\neq$	`<>`
$<$	`<`
$\leq$	`<=`
$\in$	`in`
$\subseteq$	`c=`

The first character in the Mizar representation of $\subseteq$ is the letter `c`.

1.3.3 Terms

There are three ways to write Mizar terms:

- If f is an operator symbol, which Mizar calls a *functor*, write:

  $$(x_1\,,\, x_2\,,\, \ldots,\, x_m)\ f\ (x_{m+1}\,,\, \ldots,\, x_{m+n})$$

  Again, m or n might be 0, in which case this becomes prefix or postfix notation. As an example of a postfix operator there is `^2` for square. The brackets around either list of arguments can be omitted if there is just one argument.

- If $\mathcal{L}$ and $\mathcal{R}$ are *bracket* symbols, write:

  $$\mathcal{L}\ x_1\,,\, x_2\,,\, \ldots,\, x_n\ \mathcal{R}$$

  In this case brackets `(` and `)` around the arguments are not necessary, since the symbols themselves are already brackets. An example of this kind of term is the ordered pair `[x,y]`. In that case $n = 2$, $\mathcal{L}$ is `[` and $\mathcal{R}$ is `]`.

- Any natural number is a Mizar term. If you write natural numbers, you should add to your `environ` the line:

  ```
  requirements SUBSET, NUMERALS, ARITHM;
  ```

  because else they won't behave like natural numbers.

The word *functor* for function symbol or operator symbol is Mizar terminology. It has nothing to do with the notion of functor in category theory. It is used to distinguish a function symbol in the logic from a function object in the set theory (which is a set of pairs).

Again to find out what operators are available to you, you will need to browse the MML. Here is a short list of some of the most frequent used operators to get you started.

$\emptyset$	`{}`
$\{x\}$	`{` x `}`
$\{x, y\}$	`{` x`,`y `}`
$X \cup Y$	X `\/` Y
$X \cap Y$	X `/\` Y
$\mathbb{N}$	`NAT`
$\mathbb{Z}$	`INT`
$\mathbb{R}$	`REAL`
$x + y$	x `+` y
$x - y$	x `-` y
$-x$	`-` x
xy	x `*` y
x/y	x `/` y
x^2	x`^2`
x^y	x `to_power` y
$\sqrt{x}$	`sqrt` x
$\mathcal{P}(X)$	`bool` X
(x, y)	`[`x`,`y`]`
$X \times Y$	`[:`X`,`Y`:]`
Y^X	`Funcs(`X`,`Y`)`
$f(x)$	f`.`x
$\langle\rangle$	`<*>`D
$\langle x\rangle$	`<*` x `*>`
$\langle x, y\rangle$	`<*` x`,`y `*>`
$p \cdot q$	p`^`q

The digit `2` is part of the symbol for the square operator. The last four operators are used to build finite sequences over the set D.

The `.` operation is function application *inside the set theory*. If `f` is a symbol *in the language* representing a functor with zero left arguments and one right argument then you write:

```
f x
```

or (you can always put a term in brackets):

```
f(x)
```

If `f` is a function in the set theory – a set of pairs – then

```
f.x
```

is the image of `x` under `f`, in other words it is the `y` such that `[x,y] in f`.

Exercise 1.3.3 Translate the following Mizar formulas into mathematical notation:

```
NAT c= REAL
```

```
1/0 = 0
```

```
sqrt -1 in REAL

sqrt(x^2) <> x

{x} /\ {-x} = {x} /\ {0}

[x,y] = {{x,y},{x}}

p = <*p.1,p.2*>
```

Exercise 1.3.4 Translate the following mathematical formulas into Mizar syntax:

$\sqrt{xy} \leq \frac{x+y}{2}$

$(-1, \sqrt{2}) \in \mathbb{Z} \times \mathbb{R}$

$X \cap Y \subseteq X \cup Y$

$Y^X \in \mathcal{P}(\mathcal{P}(X \times Y))$

$\langle x, y \rangle = \langle x \rangle \cdot \langle y \rangle$

$p \cdot \langle\rangle = p$

$f(g(x)) \neq g(f(x))$

1.3.4 Types: modes and attributes

The one thing left for you to understand to write Mizar formulas is Mizar's types. Although Mizar is based on ZF-style set theory – so the *objects* of Mizar are untyped – the *terms* of Mizar are typed.

An example of a Mizar type is:

```
non empty finite Subset of NAT
```

This type should be parsed as:

A Mizar type consists of an instance of a *mode* with in front a cluster of *adjectives*. The type without the adjectives is called the *radix type*. In this case the mode is `Subset` and its argument is `NAT`, the radix type is `Subset of NAT`, and the two adjectives are `non empty` and `finite`.

To put this abstractly, a Mizar type is written:

$$\alpha_1 \ \alpha_2 \ \ldots \ \alpha_m \ \mathcal{M} \ \texttt{of} \ x_1, x_2, \ldots, x_n$$

where α_1, ... α_m are adjectives, $\mathcal{M}$ is a mode symbol and x_1, x_2, ..., x_n are terms. The keyword `of` binds the arguments of the mode to the mode. It's like the brackets in a term.

The number of arguments n is part of the definition of the mode. For instance for `set` it is zero. You can't write `set of` ..., because there does not exist a `set` mode that takes arguments.

Modes are dependent types. To find out what modes are available to you, you will need to browse the MML. Here is a short list of some of the most frequent used modes to get you started.

`set`
`number`
`Element of` X
`Subset of` X
`Nat`
`Integer`
`Real`
`Ordinal`
`Relation`
`Relation of` X`,`Y
`Function`
`Function of` X`,`Y
`FinSequence of` X

Note that there both are modes `Relation` and `Function` with no arguments, and more specific modes `Relation of X,Y` and `Function of X,Y` with two arguments. They share the mode symbol but they are different.

Also note that the modes depend on *terms*. So there are no types representing the functions from a type to a type. The `Function` mode represents the functions from a set to a set. As an example the function space $(X \to Y) \times X \to Y$ corresponds to the Mizar type `Function of [:Funcs(X,Y),X:],Y`

Adjectives restrict a radix type to a subtype. They either are an *attribute*, or the negation of an attribute using the keyword `non`. Again, to find out what attributes are available to you, you will need to browse the MML. Here is a list of a few attributes.

`empty`
`even`
`odd`
`prime`
`natural`
`integer`
`real`
`finite`
`infinite`
`countable`

Maybe you now wonder about how types are used in Mizar. To clarify this take a look at three Mizar notions – `NAT`, `Nat` and `natural` – that all mean the same thing: the set of natural numbers. Here is a table that compares their uses:

meaning	*declaration*	*formula*
$n \in \mathbb{N}$		`n in NAT`
$n : \mathbb{N}$	`n be Nat`	`n is Nat`
$n : \mathbb{N}$	`n be natural number`	`n is natural`

`NAT` is a term, `Nat` is a type and `natural` is an adjective. `be`/`being` are a typing in a declaration and go between a variable and a type. `in` is the $\in$ relation of the set theory and goes between two terms. `is` goes between a term and a type or between a term and an adjective. Note that in Mizar you can have a 'type judgement' as a formula in the logic.

The example We now give the statement of the theorem for the example. The statement that we will prove in this chapter is:

```
reserve a,b,c,m,n for Nat;

a^2 + b^2 = c^2 & a,b are_relative_prime & a is odd implies
 ex m,n st m <= n & a = n^2 - m^2 & b = 2*m*n & c = n^2 + m^2;
```

So put this after the `begin` line of your `my_mizar.miz` file.

We also could have made the quantification of the `a`, `b` and `c` explicit:

```
for a,b,c st
 a^2 + b^2 = c^2 & a,b are_relative_prime & a is odd holds
  ex m,n st m <= n & a = n^2 - m^2 & b = 2*m*n & c = n^2 + m^2;
```

but it is not necessary. Mizar automatically quantifies over free variables.

We now analyze this statement in some detail. The formula has the structure:

$$\phi_1 \texttt{ \& } \phi_2 \texttt{ \& } \phi_3 \texttt{ implies ex m,n st } \phi_4 \texttt{ \& } \phi_5 \texttt{ \& } \phi_6 \texttt{ \& } \phi_7\texttt{;}$$

The predicate logic formula that corresponds to this is:

$$(\phi_1 \wedge \phi_2 \wedge \phi_3) \Rightarrow \exists\, m, n.\, (\phi_4 \wedge \phi_5 \wedge \phi_6 \wedge \phi_7)$$

The first three atomic formulas in this are:

$$\begin{aligned} \phi_1 &\equiv \texttt{a\^{}2 + b\^{}2 = c\^{}2} \\ \phi_2 &\equiv \texttt{a,b are\_relative\_prime} \\ \phi_3 &\equiv \texttt{a is odd} \end{aligned}$$

They have the structure:

$$\begin{aligned} \phi_1 &\equiv t_1 \boxed{\texttt{=}}\, t_2 \\ \phi_2 &\equiv t_3\texttt{,}\, t_4 \boxed{\texttt{are\_relative\_prime}} \\ \phi_3 &\equiv t_5 \texttt{ is } \mathcal{T} \end{aligned}$$

In this `=` and `are_relative_prime` are predicate symbols. $\mathcal{T}$ is the adjective `odd`. So we here see both kinds of atomic formula: twice a relation between terms and once a typing.

The first term t_1 in ϕ_1 is:

$$t_1 \equiv \texttt{a\^{}2 + b\^{}2}$$

It has the structure:

$$\begin{aligned} t_1 &\equiv u_1 \boxed{\texttt{+}} u_2 \\ u_1 &\equiv v_1 \boxed{\texttt{\^{}2}} \\ u_2 &\equiv v_2 \boxed{\texttt{\^{}2}} \\ v_1 &\equiv \texttt{a} \\ v_2 &\equiv \texttt{b} \end{aligned}$$

In this `+` and `^2` are functor symbols.

Exercise 1.3.5 Find Mizar types for the following concepts:

odd number that is not a prime number

empty finite sequence of natural numbers

uncountable set of reals

element of the empty set

non-empty subset of the empty set

What is the problem with the last two concepts? Do you think they should be allowed in Mizar? Study this manual to find out whether they are.

Exercise 1.3.6 Write the following statements as Mizar formulas:

The only even prime number is 2.

If a prime number divides a product it divides one of the factors.

There is no biggest prime number.

There are infinitely many prime twins.

Every even number ≥ 4 is the sum of two primes.

Write these statements first using `reserve` statements. Then write them again but this time with the types in the formulas.

1.4 Step 4: getting the environment right

Add the statement that you wrote to your article. Then check it. You will get error messages:

```
environ
 vocabularies MY_MIZAR;

begin
 reserve a,b,c,m,n for Nat;
::>                      *151

a^2 + b^2 = c^2 & a,b are_relative_prime & a is odd implies
::>,203
 ex m,n st m <= n & a = n^2 - m^2 & b = 2*m*n & c = n^2 + m^2;

::> 151: Unknown mode format
::> 203: Unknown token
```

That's because you didn't import what you used from the MML. There's nothing wrong with the statement, there's something wrong with the environment (it's empty). To correct this, you need to add *directives* to the `environ` part of the article to import what you need.

It is hard to get the environment of a Mizar article correct. In practice people often just copy the environment of an old article. However that doesn't help much when it doesn't work, and occasionally it doesn't work. So you still will have to understand how the environment works, to get it right when you get environment related errors.

Exercise 1.4.1 Try to use an environment of an existing article from the MML. Do the errors go away? If so, you might consider the rest of *Step 4* for now, and just use this environment.

1.4.1 Vocabulary, notations, constructors

The rule is quite simple. For everything you use – predicate, functor, mode or attribute – you have to add a relevant reference to three directives:

- `vocabularies`
- `notations`
- `constructors`

The list of references for `notations` and `constructors` is generally almost identical. In fact, if you follow the algorithm from this section to get them right they *will* be identical. These directives are about:

- Lexical analysis. The tokens in a Mizar article come from lists called *vocabularies*. Mizar variables are identifiers with a fixed syntax, but the

predicates, functors, types and attributes all are *symbols* which can contain any character. You need to indicate from what vocabularies you use symbols.

- Parsing of expressions. To have an expression you need to list the articles that you use predicates, functors, types and attributes from. The `notations` directive is for the syntax of expressions. The `constructors` directive is for its meaning.

Here is a list of what was use in the statement, what kind of thing it is, and what vocabularies and articles you need for it:

symbol	*kind*	*vocabulary*	*article*
=	*pred*		HIDDEN
<=	*pred*	HIDDEN	XREAL_0
+	*func*	HIDDEN	XCMPLX_0
*	*func*	HIDDEN	XCMPLX_0
-	*func*	ARYTM_1	XCMPLX_0
Nat	*mode*	HIDDEN	NAT_1
are_relative_prime	*pred*	ARYTM_3	INT_2
^2	*func*	SQUARE_1	SQUARE_1
odd	*attr*	MATRIX_2	ABIAN

You don't need to refer to the `HIDDEN` vocabulary or the `HIDDEN` article but you need to list the others. The vocabularies should go in the `vocabularies` directive and the articles should go both in the `notations` and `constructors` directives. So your environment needs to be:

```
environ
 vocabularies MY_MIZAR, ARYTM_1, ARYTM_3, SQUARE_1, MATRIX_2;
 notations XREAL_0, XCMPLX_0, NAT_1, INT_2, SQUARE_1, ABIAN;
 constructors XREAL_0, XCMPLX_0, NAT_1, INT_2, SQUARE_1, ABIAN;
```

Here is the way to find out what the vocabulary and article a given symbol are:

- To find the vocabulary use the command:

  `findvoc -w '`*symbol*`'`

  For instance the command:

  ```
  findvoc -w '-'
  ```

  gives:

  ```
  FindVoc, Mizar Ver. 7.0.01 (Win32/FPC)
  Copyright (c) 1990,2003 Association of Mizar Users
  vocabulary: ARYTM_1
  O-  32
  ```

The `O` means that this is a functor symbol, and 32 is the priority.

- To find the article, the easiest way is to go to the web pages of the abstracts of the MML on the Mizar web site, find a use of the symbol somewhere, and click on it to go to the definition.

Exercise 1.4.2 Find out what is in the `HIDDEN` vocabulary by typing:

```
listvoc HIDDEN
```

For each of the 25 symbols in this vocabulary, find an article that introduces a notation that uses it.

1.4.2 Redefinitions and clusters

But now things get tough. It turns out that your environment still does not work! The problem is that the types are wrong. If you check the article with the new environment you get:

```
a^2 + b^2 = c^2 & a,b are_relative_prime & a is odd implies
::> *103
 ex m,n st m <= n & a = n^2 - m^2 & b = 2*m*n & c = n^2 + m^2;
::>           *102          *103         *103           *103

::> 102: Unknown predicate
::> 103: Unknown functor
```

To get rid of this kind of typing problems is the most difficult part of writing a Mizar article. (In practice this kind of error often is caused because a *cluster* is missing. So if you are completely mystified by a Mizar typing error, start thinking 'cluster!')

The errors that you see here are indeed almost all caused by the fact that some clusters are not imported in the environment. This causes the expressions not to have the right types. For instance the first error is caused by the fact that `a^2` and `b^2` have type `Element of REAL`, while the `+` wants its arguments to have type `complex number`. We will show in detail why this is the case, and how to fix this by importing the right clusters.

A Mizar expression doesn't have just one type, it has a whole *set of types*. For instance, with the right environment the number `2` has the following types:

```
Nat
natural number
prime Nat
Integer
integer number
even Integer
Real
real number
Complex
complex number
Ordinal
ordinal number
set
finite set
non empty set
...
```

There are two ways to influence the types of an expression.

- To give a specific expression a more precise radix type, you use *redefinitions*. A functor can have many redefinitions that cause it to get different types depending on the types of its arguments.

  Here is an example of a redefinition taken from `PEPIN`:

  ```
  definition let n be Nat;
   redefine func n^2 -> Nat;
  end;
  ```

  What this does is change the type of some of the terms that use the `^2` operator.

  The original definition (of which this is a redefinition) is in `SQUARE_1`:

  ```
  definition let x be complex number;
   func x^2 -> set equals x * x;
  end;
  ```

  In `SQUARE_1` there already are two other redefinitions of `^2`:

  ```
  definition let x be Element of COMPLEX;
   redefine func x^2 -> Element of COMPLEX;
  end;
  definition let x be Element of REAL;
   redefine func x^2 -> Element of REAL;
  end;
  ```

  Now suppose that in your environment you have the directive:

```
notations ..., SQUARE_1, PEPIN, ...;
```

Then if you write an expression t`^2`, the type of this expression will depend on the type of t. All definitions and redefinitions of `^2` will be considered in the order that they are imported through the `notations` directive. So in this case the definition for `complex number` is first, then there is the redefinition for `Element of COMPLEX`, then the redefinition for `Element of REAL`, and finally (go to the next article, which is `PEPIN`) there is the redefinition for `Nat`.

Now the rule is that the *last* redefinition that fits the type of all arguments applies.

So if t has type `Nat` then t`^2` has type `Nat` too, while if t does not have any of the types `Element of COMPLEX` or `Element of REAL` or `Nat`, then it will have type `set`.

Note that since the the order of the articles in the `notations` directive is important for this!

- To generate more adjectives for an expression, Mizar has something called *clusters*. The process of adding adjectives according to these clusters is called the *rounding up* of clusters.

  Here are three examples of clusters, taken from `FINSET_1`.

```
registration
 cluster empty -> finite set;
end;
```

  This means that every `set` that has adjective `empty` also will get adjective `finite`.

```
registration let B be finite set;
 cluster -> finite Subset of B;
end;
```

  This means that every expression that has type `Subset of B` where `B` has type `finite set` also will get adjective `finite`.

```
registration let X,Y be finite set;
 cluster X \/ Y -> finite;
end;
```

  This means that every expression of the shape `X \/ Y` where `X` and `Y` have type `finite set` also will get adjective `finite`.

  These examples show both kinds of cluster that add adjectives to the set of types of an expression. The first two do this based on the *type* of the expression (this is called 'rounding up' a type), and the third add adjectives based on the *shape* of the expression.

To summarize: redefinitions are for narrowing the radix type and clusters are for extending the set of adjectives. (There are also redefinitions that have nothing to do with typing – because they redefine something different from the type – and a third kind of cluster that has nothing to do with adding adjectives. You should not be confused by this.)

You can always test whether some type $\mathcal{T}$ is in the set of types of an expression t by changing it to `(`t `qua` $\mathcal{T}$`)`. You get an error if t didn't have $\mathcal{T}$ in its set of types. In that case you might start looking for a redefinition or cluster that changes this situation.

The example The first error in the example:

```
a^2 + b^2 = c^2 & a,b are_relative_prime & a is odd implies
::> *103
 ex m,n st m <= n & a = n^2 - m^2 & b = 2*m*n & c = n^2 + m^2;
::>           *102          *103         *103           *103

::> 102: Unknown predicate
::> 103: Unknown functor
```

is that the `+` in `a^2 + b^2` is not well-typed. (The `*` of the error message `*103` indicates the position of the error. In this case it is below the `+` symbol.) The definition of `^2` that is in effect here is from `SQUARE_1`:

```
definition let x be Element of REAL;
 redefine func x^2 -> Element of REAL;
end;
```

and the definition of `+` is from `XCMPLX_0`:

```
definition let x,y be complex number;
 func x+y means ...
 ...
end;
```

So this shows that you would like `a^2` and `b^2` which have type `Element of REAL` to also get type `complex number`. This means that you want these expressions to get an extra adjective – the adjective `complex` – and so you need a cluster. (Once again: for more adjectives you need clusters, while for a more precise radix type you need a redefinition.)

It turns out that an appropriate cluster is in `XCMPLX_0`:

```
registration
 cluster -> complex Element of REAL;
end;
```

After you add

```
registrations XCMPLX_0;
```

to the environment the first error indeed is gone and you only have two errors left:

```
a^2 + b^2 = c^2 & a,b are_relative_prime & a is odd implies
 ex m,n st m <= n & a = n^2 - m^2 & b = 2*m*n & c = n^2 + m^2;
::>           *102                        *103

::> 102: Unknown predicate
::> 103: Unknown functor
```

The next error (the `*102` below the `<=`) is similar to the previous one. The `<=` predicate expects arguments of type `real number`, but `m` and `n` have type `Nat`. If you study the MML for some time you will find that `Nat` is the same as `Element of omega` (the definition of `Nat` in `NAT_1` and the synonym in `NUMBERS`.)

Therefore the following two clusters give you what you need. From `ARYTM_3`:

```
registration
 ...
 cluster -> natural Element of omega;
end;
```

and from `XREAL_0`:

```
registration
 cluster natural -> real number;
 ...
end;
```

The cluster directive now has become:

```
registrations XCMPLX_0, ARYTM_3, XREAL_0;
```

With this directive in the environment the only error left is:

```
a^2 + b^2 = c^2 & a,b are_relative_prime & a is odd implies
 ex m,n st m <= n & a = n^2 - m^2 & b = 2*m*n & c = n^2 + m^2;
::>                                        *103

::> 103: Unknown functor
```

This error is caused by the fact that Mizar does not consider `2` to be a `Nat`. You will see below, after the discussion of the `requirement` directive, how to get rid of this final error.

1.4.3 The other directives

Here is a list of the eight kinds of directives in the `environ` and the kind of reference they take:

`vocabularies`	*vocabulary*
`notations`	*article*
`constructors`	*article*
`registrations`	*article*
`definitions`	*article*
`theorems`	*article*
`schemes`	*article*
`requirements`	`BOOLE`, `SUBSET`, `NUMERALS`, `ARITHM`, `REAL`

Here is when you need the last four kinds of directive:

- The `definitions` directive is *not* about being able to use the theorems that are part of the definitions (that's part of the `theorems` directive.) It's about automatically unfolding predicates in the thesis that you are proving.

  This directive is useful but not important. You can ignore it until you get up to speed with Mizar.

- The `theorems` and `schemes` directives list the articles you use theorems and schemes from. So whenever you refer to a theorem in a proof, you should check whether the article is in the list of this directive.

  These are easy directives to get right.

- The `requirements` directive makes Mizar know appropriate things automatically.

  For instance to give numerals type `Nat` you need

  ```
  requirements SUBSET, NUMERALS;
  ```

  This is the solution to the last typing error left in your article.

  With `ARITHM` Mizar also knows some basic equations about numbers automatically. For instance with it, Mizar accepts `1+1 = 2` without proof.

  This is an easy directive to get right. Just put in all requirements and be done with it.

So now that you got the environment right, the article checks like this:

```
environ
 vocabularies MY_MIZAR, ARYTM_1, ARYTM_3, SQUARE_1, MATRIX_2;
 notations XREAL_0, XCMPLX_0, NAT_1, INT_2, SQUARE_1, ABIAN;
 constructors XREAL_0, XCMPLX_0, NAT_1, INT_2, SQUARE_1, ABIAN;
 registrations XCMPLX_0, ARYTM_3, XREAL_0;
 requirements SUBSET, NUMERALS;

begin
 reserve a,b,c,m,n for Nat;

a^2 + b^2 = c^2 & a,b are_relative_prime & a is odd implies
```

```
 ex m,n st m <= n & a = n^2 - m^2 & b = 2*m*n & c = n^2 + m^2;
::>                                                          *4

::> 4: This inference is not accepted
```

The only error left is `*4`. It means that Mizar is not able to prove the statement on its own.

This is an important milestone. Articles with only `*4` errors are 'almost finished'. They just need a bit of proof.

Exercise 1.4.3 For each of the following statements find an environment that makes all errors different from error `*4` go away:

```
for X,Y being non empty set,
 f being Function of X,Y, g being Function of Y,X st
  f is one-to-one & g is one-to-one holds
  ex h being Function of X,Y st h is bijective

for p being FinSequence, D being set st
 for i being Nat st i in dom p holds p.i in D holds
 p is FinSequence of D;

for G being Group, H being Subgroup of G st
 G is finite holds ord G = ord H * index H;

for GX being TopSpace, A,C being Subset of GX st
 C is connected & C meets A & C \ A <> {}GX holds
 C meets Fr A;
```

Exercise 1.4.4 Consider the nine types:

```
Element of NAT
Element of INT
Element of REAL
Element of COMPLEX
Nat
Integer
Real
Complex
natural number
integer number
real number
complex number
```

There are 81 pairs of different types $\mathcal{T}_1$ and $\mathcal{T}_2$ from this list for which the formula:

`for x being` $\mathcal{T}_1$ `holds (x qua` $\mathcal{T}_2$`) = x;`

should be allowed (however not all of them are provable in the MML). What is an environment that gives the minimum number of error messages `*116` (meaning `Invalid "qua"`)? What is used from the articles in this environment?

Exercise 1.4.5 Apart from the kinds of cluster that we showed in Section 1.4.2 (the ones that generate extra adjectives for terms), Mizar has something called *existential clusters*. These are the clusters without `->`. They don't generate extra adjectives for a term. Instead they are needed to be allowed to add adjectives to a type.

The reason for this is that Mizar types always have to be non-empty. So to be allowed to use a type something has to be proved. That is what an existential cluster does.

For example to be allowed to use the type:

```
non empty finite Subset of NAT
```

you needs an existential cluster from `GROUP_2`:

```
registration let X be non empty set;
 cluster finite non empty Subset of X;
end;
```

If you don't have the right existential clusters in your article you will get error `*136` which means `non registered cluster`.

Which types in the following list have an existential cluster in the MML? For the ones that have one, where did you find them? For the ones that don't have one, what would be an appropriate existential cluster?

```
empty set
```

```
odd prime Nat
```

```
infinite Subset of REAL
```

```
non empty Relation of NAT,NAT
```

1.5 Step 5, maybe: definitions and lemmas

Step 5 you can skip. Then you just start working on the proof of the theorem immediately.

However if you did step 1 right, you probably have some lemmas that you know you will need. Or maybe you need to define some notions before being able to state your theorem at all.

So do so in this step. Add relevant definitions and lemmas to your article now. Just like the statement in step 3. Without proof. Yet.

1.5.1 Functors, predicates

There are two ways to define a functor in Mizar:

- As an abbreviation:

```
definition let x be complex number;
 func x^2 -> complex number equals x * x;
 coherence;
end;
```

The `coherence` correctness condition is that the expression `x * x` really has type `complex number` like it was claimed. If Mizar can't figure this out by itself you will have to prove it.

- By a characterization of the result:

```
definition let a be real number;
 assume 0 <= a;
 func sqrt a -> real number means
  0 <= it & it^2 = a;
 existence;
 uniqueness;
end;
```

In the characterizing statement the variable `it` is the result of the functor.

The `existence` and `uniqueness` are again correctness conditions. They state that there always exists a value that satisfies the defining property, and that this value is unique. (In the proof of them you are allowed to use that the assumption from the definition is satisfied.)

In this case `existence` is the statement:

```
ex x being real number st 0 <= x & x^2 = a
```

and `uniqueness` is the statement:

```
for x,x' being real number st
 0 <= x & x^2 = a & 0 <= x' & x'^2 = a holds x = x'
```

And here is an example of the way to define a predicate:

- ```
  definition let m,n be Nat;
   pred m,n are_relative_prime means
    m hcf n = 1;
  end;
  ```

  The definition of a predicate doesn't have a correctness condition. (In this example `hcf` is the greatest common divisor.)
  ```

1.5.2 Modes and attributes

There are two ways to define a mode in Mizar:

- As an abbreviation:

```
definition let X,Y be set;
 mode Function of X,Y is
  quasi_total Function-like Relation of X,Y;
end;
```

  In this definition `quasi_total` and `Function-like` are attributes.

- By a characterization of its elements.

```
definition let X,Y be set;
 mode Relation of X,Y means
  it c= [:X,Y:];
 existence;
end;
```

  The `existence` correctness condition states that the mode is inhabited. This has to be the case because Mizar's logic wants all Mizar types to be non-empty.

  In this case `existence` is the statement:

```
ex R st R c= [:X,Y:]
```

And here is an example of the way to define an attribute:

- ```
  definition let i be number;
   attr i is even means
    ex j being Integer st i = 2*j;
  end;
  ```

The example The essential lemma for the example is that if two numbers are relative prime and their product is a square, then they are squares themselves. In Mizar syntax this is the statement:

```
m*n is square & m,n are_relative_prime implies
 m is square & n is square
```

Now the attribute `square` is already present in the MML, but it is in the article `PYTHTRIP`, and that is the article that you are currently writing! So let's prentend it is not there and add it to the article, to practice writing definitions.

The way to define the attribute `square` (this will enable you to have a type `square number`) is:
```

```
definition let n be number;
 attr n is square means
  ex m being Nat st n = m^2;
end;
```

(Note that we do not only define the attribute `square` for expressions of type `Nat`, but for all Mizar terms. Else statements like `not -1 is square` would not be well typed.)

Part of most definitions is the introduction of a new symbol. In this case it is the attribute symbol `square`. Again, it is already in the MML, in vocabulary `PYTHTRIP`. But again, we do not want to use that, so let's add it to the vocabulary of the `MY_MIZAR` that you are writing.

So add the line:

```
Vsquare
```

to the `my_mizar.voc` file.

The `V` in front means that this is an attribute symbol. Vocabularies have an `O` in front for functors, `R` in front for predicates, `M` in front for modes and `V` in front for attributes.

The statements in a Mizar article can be local to the article or be visible to the outside of the article. In the latter case they have to be preceded by the keyword `theorem`. So add `theorem` in front of the statements, like this:

```
theorem Th1:
 m*n is square & m,n are_relative_prime implies
  m is square & n is square;
::>                        *4,4

theorem Th2:
 a^2 + b^2 = c^2 & a,b are_relative_prime & a is odd implies
  ex m,n st m <= n & a = n^2 - m^2 & b = 2*m*n & c = n^2 + m^2;
::>                                                        *4

::> 4: This inference is not accepted
```

(Note that there are *two* `*4` errors below theorem `Th1`. The reason for this is that the Mizar checker considers the two conclusions `m is square` and `n is square` to be two different proof obligations.)

Exercise 1.5.1 Show that in Mizar definition by abbreviation is secondary to definition by characterization:

How can you simulate the `equals` variant of a `func` definition with the `means` variant of a `func` definition? Similarly: how can you simulate the `is` variant of a `mode` definition with the `means` variant of a `mode` definition?

Exercise 1.5.2 We claimed that Mizar's types are always non-empty. However you can write the type:

```
Element of {}
```

Explain why this is not a problem.

1.6 Step 6: proof skeleton

Mizar is a block structured language like the Pascal programming language. Instead of `begin`/`end` blocks it has `proof`/`end` blocks. And instead of procedures it has theorems. Apart from that it is a rather similar language.

We will now write the proof of the theorem. For the moment we just write the steps and not link them together. Linking the steps in the proof will be step 7 of the nine easy steps.

A Mizar proof begins with the keyword `proof` and ends with the keyword `end`.

Important! If you have a proof after a statement there should not be a semicolon after the statement. So:

statement `;`
`proof`
proof steps
`end;`

is wrong! It should be:

statement
`proof`
proof steps
`end;`

This is easy to do wrong. Then you will get a `*4` error at the semicolon that shouldn't be there.

In this section we will discuss the following kinds of proof step. There are some more but these are the frequently used ones:

step	*section*
compact	1.6.2
`assume`	1.6.1
`thus`	1.6.1
`let`	1.6.1
`take`	1.6.1
`consider`	1.6.2
`per cases/suppose`	1.6.2
`set`	1.6.3
`reconsider`	1.6.3
iterative equality	1.6.4

1.6.1 Skeleton steps

During the steps of the proof the statement that needs to be proved is kept track of. This is the 'goal' of the proof. It is called the *thesis* and can be referred to with the keyword `thesis`.

A Mizar proof consists of steps. Steps contain statements that you know to be true in the given context. Some of the steps reduce the thesis. At the `end` of the proof the thesis should be reduced to $\top$ or else you will get error `*70` meaning `Something remains to be proved`. Steps that change the thesis are called *skeleton steps*.

The four basic skeleton steps are `assume`, `thus`, `let` and `take`. They correspond to the shape of the thesis in the following way:

thesis to be proved, before	*the step*	*thesis to be proved, after*
ϕ `implies` ψ	`assume` ϕ	ψ
ϕ `&` ψ	`thus` ϕ	ψ
`for x being` $\mathcal{T}$ `holds` ψ	`let x be` $\mathcal{T}$	ψ
`ex x being` $\mathcal{T}$ `st` ψ	`take` t	$\psi[\mathtt{x} := t]$

Just like with `for` the typing can be left out of the `let` if the variable is in a `reserve` statement.

Here is an example of how these skeleton steps work in a very simple proof:

```
for x,y st x = y holds y = x
proof
 let x,y;
 assume x = y;
 thus y = x;
end;
```

At the start of the proof the thesis is `for x,y st x = y holds y = x`. After the `let` step it is reduced to `x = y implies y = x`. After the `assume` step it is reduced to `y = x`. After the `thus` step it is reduced to $\top$ and the proof is complete.

The skeleton steps of the proof of the example will be:

```
theorem Th2:
 a^2 + b^2 = c^2 & a,b are_relative_prime & a is odd implies
  ex m,n st m <= n & a = n^2 - m^2 & b = 2*m*n & c = n^2 + m^2
proof
 assume a^2 + b^2 = c^2;
 assume a,b are_relative_prime;
 assume a is odd;
 take m,n;
 thus m <= n;
 thus a = n^2 - m^2;
 thus b = 2*m*n;
 thus c = n^2 + m^2;
end;
```

At this point we don't have anything to put in the place of the m and n. We need another kind of step for that.

Exercise 1.6.1 Explain why you can always end a proof with a `thus thesis` step. This is the Mizar version of QUOD ERAT DEMONSTRANDUM. Count in the MML how many times this construction occurs. (Include in your count `hence thesis` which is the same thing.) Is there an occurence of `thesis` in the MML which is *not* part of this construction?

Exercise 1.6.2 Write the skeleton steps in the proofs for the following Mizar statements:

```
x = 0 & y = 0 implies x + y = 0 & x*y = 0

ex x st x in X implies for y holds y in X

for n st not ex m st n = 2*m holds ex m st n = 2*m + 1

(ex n st n in X) implies
 ex n st n in X & for m st m < n holds not m in X
```

Write a fresh variable in the `take` steps if there is no good choice for the term.

1.6.2 Compact statements and elimination

The simplest step is a *compact step*. This is just a statement that is true in the current context. This is the most common step in a Mizar proof. So a Mizar proof skeleton for the most part just looks like:

```
...
statement;
statement;
statement;
...
```

where each statement in the list is a consequence of a combination of some of the preceding statements.

And then there are the `consider` and `per cases`/`suppose` constructions. They are both related to a statement:

if you can prove this	...	*you can have this as a proof step*
`ex x st` ϕ		`consider x such that` ϕ`;`

ϕ_1 `or` ϕ_2 `or ... or` ϕ_n		`per cases;`
		`suppose` ϕ_1`;`
		proof for the case ϕ_1
		`end;`
		`suppose` ϕ_2`;`
		proof for the case ϕ_2
		`end;`
		...
		`suppose` ϕ_n`;`
		proof for the case ϕ_n
		`end;`

To use natural deduction terminology, these steps correspond to existential and disjuction *elimination*. The `consider` step introduces a variable that you can refer to in your proof, just like the `let` step.

We can now add the following steps to the example proof to give us terms `m` and `n` for the `take` step:

```
...
((c - a)/2)*((c + a)/2) = (b/2)^2;
((c - a)/2)*((c + a)/2) is square;
(c - a)/2,(c + a)/2 are_relative_prime;
(c - a)/2 is square & (c + a)/2 is square;
consider m such that (c - a)/2 = m^2;
consider n such that (c + a)/2 = n^2;
take m,n;
...
```

The first four steps are compact statements. The two `consider` steps use the existential statement that's part of the definition of `square`.

1.6.3 Macros and casts

Often in proofs certain expressions occur many times. It is possible to give such expressions a name using `set` commands:

```
set h = b/2;
set m2 = (c - a)/2;
set n2 = (c + a)/2;
```

A `set` command is a definition of a constant that is local to the proof. It behaves very much like a *macro*.

When writing Mizar articles, often you want to use an expression with a different type than it's got. You can change the type of an expression with the `reconsider` command. This is like a `set`, but also gives the type of the new variable.

In the example we will need the abbreviated variables `h`, `m2` and `n2` to have type `Nat`. This can be accomplished by changing the `set` lines in the previous paragraph to:

```
reconsider h = b/2 as Nat;
reconsider m2 = (c - a)/2 as Nat;
reconsider n2 = (c + a)/2 as Nat;
```

Again the new variables behave like macros, but now with a different type. So `reconsider` is a way to *cast* the type of a term.

1.6.4 Iterative equalities

In mathematical calculations one often has a chain of equalities. Mizar has this feature too.

You can write a calculation:

$$\left(\frac{c-a}{2}\right)\left(\frac{c+a}{2}\right) = \frac{(c-a)(c+a)}{4} = \frac{c^2-a^2}{4} = \frac{b^2}{4} = \left(\frac{b}{2}\right)^2$$

in Mizar as:

```
((c - a)/2)*((c + a)/2) = (c - a)*(c + a)/4
 .= (c^2 - a^2)/4
 .= b^2/4
 .= (b/2)^2;
```

Such a chain of `.=` equalities is called an *iterative equality*. Note that the first equality in the chain is written with the `=` symbol instead of with the `.=` symbol.

Exercise 1.6.3 Collect the steps for the proof of `Th2` that were given in this section. Put them in the proper order in your `my_mizar.miz` file. Use the `reconsider` lines (not the `set` lines) and replace the abbreviated expressions everywhere by their abbreviation. Put in the iterated equality as well.

Add two compact statements before the `reconsider` lines stating that `b` is even and that `c` is odd. They will be needed to justify the `reconsider` lines. Add two compact statements relating `m2` and `n2` to `a` and `c` after the `reconsider` lines.

Now check your file with `mizf`. Update the environment if necessary until you have only `*4` errors left. How many `*4` errors do you get?

1.7 Step 7: completing the proof

In this section you will add justifications to the steps of your proof. That will finish your article.

1.7.1 Getting rid of the *4 errors

There are three ways to justify a step in Mizar:

- By putting a semicolon `;` after it:

  statement `;`

  This is the empty justification. It tells Mizar: *figure this out by yourself.*

- By putting `by` after it with a list of labels of previous statements:

  statement `by` *reference* `,` ... `,` *reference* `;`

  In fact the previous way to justify a step is the special case of this in which the list of references is empty.

- By putting a subproof after it:

  statement `proof` *steps in the subproof* `end;`

  The subproofs are what gives Mizar proofs the block structured look.

  If you don't use `thesis` in a subproof, the statement can be reconstructed from the skeleton steps in that subproof. Therefore you might want to omit the statement. You can do this by using `now`:

  `now` *steps in the subproof* `end;`

  This is exactly the same as the statement with the subproof, but you don't need to write the statement.

Here is a small example that shows how `by` is used:

```
(x in X implies x in Y) & x in X implies x in Y
proof
 assume
A1: x in X implies x in Y;
 assume
A2: x in X;
 thus x in Y by A1,A2;
end;
```

Of course this one is so easy that Mizar can do it on its own:

```
(x in X implies x in Y) & x in X implies x in Y ;
```

Often in the list of references after a `by` justification, there is a reference to the previous statement. This can also be written by putting the keyword '`then`' in front of the step. Using `then` you can often avoid having to label statements. So you can replace:

```
A1: statement;
A2: statement;
 statement by A1,A2:
```

by

```
A1: statement;
 statement;
 then statement by A1;
```

Some people like writing the `then` on the previous line, after the statement that it refers to:

```
A1: statement;
 statement; then
 statement by A1;
```

This is a matter of taste. Other people choose the position of the `then` depending on whether the statement after the `then` has a label or not.

When you want to use `then` to justify a `thus` step you are not allowed to write `then thus`. You have to write `hence`. In Mizar when in certain combinations of two keyword get together you have to replace the combination by something else. Here are the two relevant equations:

```
then  +  thus  =  hence
thus  +   now  =  hereby
```

So the `hence` keyword means two things:

1. This follows from the previous step.
2. This is part of what was to be proved.

A common Mizar idiom is `hence thesis`. You see that many times in any Mizar article.

1.7.2 Properties and requirements

So now you need to hunt the MML for relevant theorems. This is made difficult by three reasons:

- The MML is big.
- The MML is not ordered systematically but chronologically. It is like a mediaeval library in which the books are put on the shelf in the order that they have been acquired.
- Mizar is smart. The theorem you can use might not look very much like the theorem you are looking for.

As an example of the fact that theorems can be in unexpected places, the basic theorem that relates the two kinds of dividability that are in the Mizar libary:

```
m divides n iff m divides (n qua Integer);
```

is in an article called `SCPINVAR` which is about loop invariants of a certain small computer program (SCP stands for SCMPDS program and SCMPDS stands for small computer model with push-down stack).

The best way to find theorems in the MML is by using the `grep` command to search all the `.abs` abstract files. Although the MML is huge it's small enough for this to be practical. For instance this is an appropriate command to search for the theorem in the `SCPINVAR` article:

```
% grep 'divides .* qua' $MIZFILES/abstr/*.abs
/usr/local/lib/mizar/abstr/scpinvar.abs:   m divides n iff m divides (n qua Integer);
%
```

Another problem when looking for theorems is that Mizar is too smart.

If you look at the definitions of `+`, `*` and `<=` you will find that they contain the keywords `commutativity`, `reflexivity`, `connectedness`, `synonym` and `antonym`:

```
definition let x,y be complex number;
 func x + y means
:: XCMPLX_0:def 4
  ...
  commutativity;
 func x * y means
:: XCMPLX_0:def 5
  ...
  commutativity;
end;
```

```
definition let x,y be real number;
 pred x <= y means
:: XREAL_0:def 2
  ...
  reflexivity;
  connectedness;
end;
```

```
notation let x,y be real number;
  synonym y >= x for x <= y;
  antonym y < x for x <= y;
  antonym x > y for x <= y;
end;
```

These keywords mean two things:

- If you use a `synonym` or `antonym` then Mizar will internally use the real name. So if you write:

  ```
  a < b
  ```

  then internally Mizar will consider this to be an abbreviation of:

  ```
  not b <= a
  ```

  and if you write:

  ```
  a >= b
  ```

  then internally Mizar will consider this to mean:

  ```
  b <= a
  ```

- Mizar will always 'know' when justifying a step, about the properties `commutativity`, `reflexivity` and `connectedness`. So if you have a statement:

  ```
  ... x*y ...
  ```

  then Mizar behaves as if this statement is exactly the same as:

  ```
  ... y*x ...
  ```

  Also Mizar will use all implication of the shape:

  $$x = y \Rightarrow x \leq y$$

  and:

  $$x \leq y \vee y \leq x$$

  when trying to do a justification. These properties mean for the $<$ relation that:

  $$\neg(x < x)$$

  and:

  $$x < y \Rightarrow x \leq y$$

  For instance if you have proved:

  ```
  a < b
  ```

  then you can refer to this when you only need:

  ```
  a <> b
  ```

This is all very nice but let's look at three examples of the subtlety it leads to:

- Suppose you want to justify the following step:

  ```
  a >= 0;
  then c - a <= c by ... ;
  ```

  It turns out that the theorem that you need is:

  ```
  theorem :: REAL_2:173
   (a<0 implies a+b<b & b-a>b) & (a+b<b or b-a>b implies a<0);
  ```

  It will give you:

  ```
  c - a > c implies a < 0
  ```

  but because of the `antonym` definition of `<` and `>` this really means:

```
not c - a <= c implies not 0 <= a
```

which is equivalent to:

```
0 <= a implies c - a <= c
```

Which is what you need.

- Suppose you need to prove the equation:

```
(c + a + c - a)/2 = (c + c + a - a)/2
```

The + and - operators have the same priority and are left associative, so this has to be read as:

```
(((c + a) + c) - a)/2 = (((c + c) + a) - a)/2
```

Because of `commutativity` of + this really is the same as:

```
((c + (c + a)) - a)/2 = (((c + c) + a) - a)/2
```

So you need to show that: `c + (c + a) = (c + c) + a` which is associativity of +, a theorem from the `XCMPLX_1` article:

```
theorem :: XCMPLX_1:1
 a + (b + c) = (a + b) + c;
```

Note that the original equation doesn't look very much like an instance of associativity!

- Consider the transitive law for <=:

```
theorem :: AXIOMS:22
 x <= y & y <= z implies x <= z;
```

It's good to have this, but where is the analogous law for <:

```
x < y & y < z implies x < z;
```

It turns out that this also is `AXIOMS:22`! To see why this is the case, note that it is a consequence of the stronger theorem:

```
x <= y & y < z implies x < z;
```

and becase of the `antonym` definition of < this is the same as:

```
x <= y & not z <= y implies not z <= x;
```

which is equivalent to:

```
z <= x & x <= y implies z <= y;
```

which is indeed `AXIOMS:22`.

(The 'arithmetical lemmas' for the arithmetical operations that we have used as examples here all are in the articles `AXIOMS`, `REAL_1`, `REAL_2` and `XCMPLX_1`. The rule is that if the lemma that your are looking for is valid in the complex numbers (which generally means that it just involves equality and not `<=` or `<`) then it is in `XCMPLX_1`. Unfortunately if it *does* talk about inequalities, then you will have to look in all the other three articles. In that case there is not an easy rule that will tell you where it has to be.)

Sometimes you should not go look for a theorem. If you have `requirements ARITHM` then Mizar knows many facts about the natural numbers without help. Sometimes those facts even don't have a `theorem` in the library anymore, so you search for a long time if you don't realize that you should use `requirements`. For instance if you use `requirements ARITHM`:

```
x + 0 = x;
```

and:

```
0*x = 0;
```

and:

```
1*x = x;
```

and:

```
0 < 1;
```

and:

```
0 <> 1;
```

all don't need any justification.

Suppose you want to justify:

```
a >= 1;
then a > 0 by ... ;
```

If you had `0 < 1` then this would just be an instance of `AXIOMS:22` (as we just saw). But `requirements ARITHM` gives that to you for free! So you can just justify this step with `AXIOMS:22`.

Exercise 1.7.1 We claim that the MML is big. Count how many source lines there are in the `.miz` articles in the MML. Try to guess how many lines of Mizar a competent Mizar author writes per day and then estimate the number of man-years that are in the MML.

1.7.3 Automation in Mizar

In Mizar it's not possible to write *tactics* to automate part of your proof effort like you can do in other proof checkers. All Mizar's automation is built into the Mizar system by its developers.

Mizar has four kinds of automation:

Semantic correlates Internally Mizar stores its formulas in some kind of conjunctive normal form that only uses $\wedge$, $\neg$ and $\forall$.

This means that it automatically identifies some equivalent formulas. These equivalence classes of formulas are called *semantic correlates*. Because of semantic correlates, the skeleton steps of a formula are more powerful than you might expect. For instance you can prove:

```
φ
proof
 assume not φ;
 ...
 thus contradiction;
end;
```

and also you can prove:

```
φ or ψ
proof
 assume not φ;
 ...
 thus ψ;
end;
```

(Mizar does not identify ϕ `&` ψ with ψ `&` ϕ. You have to put the `thus` steps in a proof in the same order as the conjuncts appear in the thesis.)

Properties and requirements Properties and requirements were discussed in the previous section.

Clusters Clusters automatically generate adjectives for expressions. Often theorems can be rephrased as clusters, after which they will become automatic.

Consider for instance the theorem:

```
m is odd square & n is odd square implies m + n is non square;
```

which says that a square can never be the sum of two odd squares. (The reason for this is that odd squares are always 1 modulo 4, so the sum would be 2 modulo 4, but an even square is always 0 modulo 4.)

This theorem can be rephrased as a cluster:

```
definition let m,n be odd square Nat;
 cluster m + n -> non square;
end;
```

Once you have proved this cluster Mizar will apply the theorem automatically when appropriate and you don't have to give a reference to it in your proofs.

Justification using `by` The `by` justifier is a weak first order prover. It tries to deduce the statement in front of the `by` from the statements that are referred to after the `by`.

A `by` justification pretty much feels like a 'natural reasoning step'. When one studies the proofs that humans write, it turns out that they tend to choose slightly smaller steps than `by` can do.

The way `by` works is:

- It puts the implication that it has to prove in conjunctive normal form. Then it tries to do each conjunct separately.

  That's why you often get more than one `*4` error for a failed justification. Mizar puts in an error for every conjunct that has not been justified.

- It then tries to prove the inference. In only *one* of the antecedents can a universal quantifier be instantiated. Therefore:

  ```
  A1: for x holds x in X;
  A2: for x holds not x in X;
   contradiction by A1,A2;
  ```

  won't work because it both has to instantiate the `for` in `A1` and `A2`. You will need to split this into:

  ```
  A1: for x holds x in X;
  A2: for x holds not x in X;
   a in X by A1;
   then contradiction by A2;
  ```

- The `by` prover will do congruence closure of all the equalities that it knows. It also will combine the typing information of equal terms.

- A conclusion that has an existential quantifier is equivalent to an antecedent that has a universal quantifier. Therefore `by` is able to derive existential statements from instantiations of it.

1.7.4 Unfolding of definitions

Here is a table that shows when Mizar will unfold definitions:

func/equals	−
func/means	−
pred/means	◇
mode/is	+
mode/means	◇
attr/means	◇
set	+

The items in this table that have a + behave like macros. For instance the definition of `Nat` is:

```
definition
 mode Nat is Element of NAT;
end;
```

If you write `Nat` it's exactly like writing `Element of NAT`. So theorems about the `Element` mode will automatically apply to `Nat`.

The items in the table that have a − will never be expanded. All you get is a definitional theorem about the notion. You will need to refer to this theorem if you want to use the definition.

The items in the table that have a ◇ will be expanded if you use the `definitions` directive. Expansion only happens in the thesis. It takes place if a skeleton step is attempted that disagrees with the shape of the thesis.

For instance the definition of `c=` is:

```
definition let X,Y;
 pred X c= Y means
:: TARSKI:def 3
  x in X implies x in Y;
 reflexivity;
end;
```

If you have `TARSKI` in your `definitions` you can prove set inclusion as follows:

```
X c= Y
proof
 let x be set;
 assume x in X;
 ...
 thus x in Y;
end;
```

Similarly consider the definition of equality in `XBOOLE_0`:

```
definition let X,Y;
 redefine pred X = Y means
:: X_BOOLE_0:def 10
  X c= Y & Y c= X;
end;
```

If you add XBOOLE_0 to your definitions it will allow you to prove:

```
X = Y
proof
 ...
 thus X c= Y;
 ...
 thus Y c= X;
end;
```

or even:

```
X = Y
proof
 hereby
  let x be set;
  assume x in X;
  ...
  thus x in Y;
 end;
 let x be set;
 assume x in Y;
 ...
 thus x in X;
end;
```

Exercise 1.7.2 Write Mizar proofs of the following three statements:

```
reserve X,Y,Z,x for set;
```

```
for X holds X c= X;
```

```
for X,Y st X c= Y & Y c= X holds X = Y;
```

```
for X,Y,Z st X c= Y & Y c= Z holds X c= Z;
```

that only make use of the following two theorems from the MML:

```
theorem :: TARSKI:2
 for X,Y holds
  X = Y iff for x holds x in X iff x in Y;
```

```
theorem :: TARSKI:def 3
 for X,Y holds
  X c= Y iff for x st x in X implies x in Y;
```

(If you look in the MML, it will turn out that in the TARSKI article this is not exactly what's there. We present it like this for didactic purposes.)

The approach you should follow in your proof, is that in order to prove a statement like X c= Z you should first prove that:

```
...
A1: for x holds x in X implies x in Z;
proof
 ...
end;
...
```

and then justify the `X c= Z` by referring to this `for` statement, together with `TARSKI:def 3`:

```
...
X c= Z by A1,TARSKI def:3;
...
```

Try to make proofs in which the argumentation is as clear as possible, but also make a second set of proofs that are as small as possible.

Exercise 1.7.3 Take the following skeleton proof of the *the drinker's principle*. It is called the drinker's principle because if `X` is the set of people in a room that are drinking, then it says that *in each room there is a person such that if that person is drinking, then everyone in that room is drinking.* This sounds paradoxical (why should one person have the power to make all other people in the room drink?) but when you really understand what it says, it is not.

```
reserve X,x,y for set;

ex x st x in X implies for y holds y in X
proof
 per cases;
 suppose ex x st not x in X;
  consider x such that not x in X;
  take x;
  assume x in X;
  contradiction;
  let y;
  thus y in X;
 end;
 suppose not ex x st not x in X;
  for x holds x in X;
  assume x in X;
  thus thesis;
 end;
end;
```

Add labels to the statements and put in justifications for all the steps.

Experiment with other proofs of the same statement. If you know constructive logic: write a Mizar proof of this statement that shows where the non-constructive steps in the proof are.

Exercise 1.7.4 Write a proof of the following statement:

```
reserve X,Y,x,y for set;

for X,Y,y holds
 ((ex x st x in X implies y in Y) &
  (ex x st y in Y implies x in X))
 iff (ex x st x in X iff y in Y);
```

Can you think of a proof that doesn't need `per cases`?

Exercise 1.7.5 Finish the proof of `Th2`. Add as many clusters and lemmas as you need. For instance you might use:

```
Lm1: m is odd square & n is odd square implies
      m + n is non square;
Lm2: n is even iff n/2 is Nat;
Lm3: m,n are_relative_prime iff
      for p being prime Nat holds not (p divides m & p divides n);
Lm4: m^2 = n^2 iff m = n;
```

but if you choose to use different lemmas that's fine too. Just put them before the theorem (you can prove them later if you like). If you don't think a lemma will be useful to others you don't have to put `theorem` in front of it.

Now add compact statements and justifications to the proof of `Th2` until all the `*4` errors are gone.

1.8 Step 8: cleaning the proof

Now that you finished your article the fun starts! Mizar has several utilities to help you improve your article. These utilities will point out to you what is in your article that is not necessary.

Those utilities don't put their error messages inside your file on their own. You need to put the name of the program as an argument to the program `revf` to accomplish that.

`relprem` This program points out references in a `by` justifications – as well occurrences of `then` – that are not necessary. These references can be safely removed from your article.

This program also points out the real errors in the article. Some people prefer to always use `relprem` instead of `mizf`.

`relinfer` This program points out steps in the proof that are not necessary. It will indicate references to statements that can be short-circuited. You can omit such a reference and replace it with the references from the step that it referred to.

As an example of `relinfer` consider the following part of a proof:

```
A1: m2*n2 = h^2 by ...;
A2: m2*n2 is square by A1;
A3: m2,n2 are_relative_prime by ...;
A4: m2 is square & n2 is square by A2,A3,Th1;
 consider m such that m2 = m^2 by A4;
 ...
```

If you run:

```
revf relinfer my_mizar
```

then you will get `*604` meaning `Irrelevant inference`:

```
A1: m2*n2 = h^2 by ...;
A2: m2*n2 is square by A1;
A3: m2,n2 are_relative_prime by ...;
A4: m2 is square & n2 is square by A2,A3,Th1;
::>                                  *604
 consider m such that m2 = m^2 by A4;
 ...
```

It means that you can replace the reference to `A2` in step `A4` with the references in the justification of `A2` itself. In this case that's `A1`. So the proof can be shortened to:

```
A1: m2*n2 = h^2 by ...;
A3: m2,n2 are_relative_prime by ...;
A4: m2 is square & n2 is square by A1,A3,Th1;
 consider m such that m2 = m^2 by A4;
 ...
```

Please take note that the `relinfer` program is dangerous! Not because your proofs will break but because they might become ugly. `relinfer` encourages you to cut steps that a human might want to see.

`reliters` (pronounced *rel-iters*, not *re-liters*) This program points out steps in an interative equality that can be skipped. If you omit such a step you have to add its references to the references of the next one.

As an example of `reliters` consider the following iterative equality:

```
 m2*n2 = (c - a)*(c + a)/(2*2) by XCMPLX_1:77
  .= (c - a)*(c + a)/4
  .= (c^2 - a^2)/4 by SQUARE_1:67
  .= b^2/4 by A1,XCMPLX_1:26
  .= b^2/(2*2)
  .= b^2/2^2 by SQUARE_1:def 3
  .= h^2 by SQUARE_1:69;
```

If you run:

```
revf reliters my_mizar
```

then you will get `*746` meaning `References can be moved to the next step of this iterative equality`:

```
 m2*n2 = (c - a)*(c + a)/(2*2) by XCMPLX_1:77
  .= (c - a)*(c + a)/4
::>                  *746
  .= (c^2 - a^2)/4 by SQUARE_1:67
  .= b^2/4 by A1,XCMPLX_1:26
::>         *746
  .= b^2/(2*2)
  .= b^2/2^2 by SQUARE_1:def 3
  .= h^2 by SQUARE_1:69;
```

So you can remove the terms with the `*746` from the iterative equality. You then have to move the references in the justification to that of the next term in the sequence. So the iterative equality can be shortened to:

```
 m2*n2 = (c - a)*(c + a)/(2*2) by XCMPLX_1:77
  .= (c^2 - a^2)/4 by SQUARE_1:67
  .= b^2/(2*2) by A1,XCMPLX_1:26
  .= b^2/2^2 by SQUARE_1:def 3
  .= h^2 by SQUARE_1:69;
```

`trivdemo` This program points out subproofs that are so simple that you can replace them with a single `by` justification. It's surprising how often this turns out to be the case!

`chklab` This program points out all labels that are not referred to. They can be safely removed from your article.

`inacc` This program points out the parts of the proofs that are not referred to. They can be safely removed from your article.

`irrvoc` This program points out all vocabularies that are not used. They can be safely removed from your article.

`irrths` This program points out the articles in the `theorem` directive that are not used for references in the proofs. They can be safely removed from your article.

Cleaning your article is fun! It really feels like polishing something after it is finished and making it beautiful.

Remember to check your article once more with `mizf` after you have finished optimizing it, to make sure that you haven't accidentally introduced any new errors.

Exercise 1.8.1 Take the proof that you made in exercise 1.7.3 and run the programs that are discussed in this section on it. Which of the programs advise you to change your proof?

Exercise 1.8.2 Consider the statements in a Mizar proof to be points in a graph and references to the statements to be the edges. What are the `relprem` and `relinfer` optimizations when you consider them as transformations of these graphs? Draw a picture!

1.9 Step 9: submitting to the library

So now your article is finished and clean. You should consider submitting it to the MML.

There are two rules that articles for the MML have to satisfy:

- It is mathematics that's not yet in the MML.

  There should not be alternate formalizations of the same concepts in the MML. There's too much double work in the MML already.

- It is significant.

  As a rule of thumb the article should be at least a 1000 lines long. If it's too short consider extending your article until it's long enough.

There are many reasons to submit your article to the MML:

- People will be able to use your work and build on it.
- Your definitions will be the standard for your subject in later Mizar articles.
- When the Mizar system changes your article will be kept compatible by the people of the Mizar group.
- An automatically generated TEX version of your article will be published in a journal called *Formalized Mathematics*.

To find the details of how to submit an article to the MML go to the web page of the Mizar project and click on the appropriate link. Be aware that to submit your article to the MML you need to sign a form that gives the Mizar people the right to change it as they see fit.

If your article is not suitable for submission to the MML but you still want to use it to write further articles, you can put it in a local library of your own. Currently this kind of local library will only support a few articles before it becomes rather inefficient. The MML has been optimized such that library files don't grow too much. You can't do this optimization for your local library yourself, because the tool for it is not distributed. So it's really better to submit your work to the MML if possible.

If you want a local library of your own there need to be a third directory called `prel` next to the already existing `dict` directory. To put the files for your article in this directory you need to run the command:

```
miz2prel text\my_mizar
```

outside your `text` and `prel` directories (it will put some files in your `prel` directory.) After you have run `miz2prel` you can then use your article in further articles.

Exercise 1.9.1 How many Mizar authors are there in the MML? Shouldn't you be one?

Chapter 2

Some advanced features

Now that you know how to write a Mizar article, let's take a look at some more advanced features of Mizar that you haven't seen yet.

To show these features we use a small proof by Grzegorz Bancerek from the article `quantal1.miz`. This article is about the mathematical notion of *quantales*. It is not important that you know what quantales are. We just use this proof to show some of Mizar's features.

```
reserve
 Q for left-distributive right-distributive complete Lattice-like
     (non empty QuantaleStr),
 a, b, c, d for Element of Q;

theorem Th5:
 for Q for X,Y being set holds "\/"(X,Q) [*] "\/"(Y,Q) = "\/"({a[*]
b: a in X & b in Y}, Q)
  proof let Q; let X,Y be set;
 deffunc F(Element of Q) = $1[*]"\/"(Y,Q);
 deffunc G(Element of Q) = "\/"({$1[*]b: b in Y}, Q);
 defpred P[set] means $1 in X;
 deffunc H(Element of Q,Element of Q) = $1[*]$2;
A1:  for a holds F(a) = G(a) by Def5;
      {F(c): P[c]} = {G(a): P[a]} from FRAENKEL:sch 5(A1);
   hence
    "\/"(X,Q) [*] "\/"(Y,Q) =
    "\/"({"\/"({H(a,b) where b is Element of Q: b in Y}, Q)
       where a is Element of Q: a in X}, Q) by Def6 .=
    "\/"({H(c,d) where c is Element of Q,
                  d is Element of Q: c in X & d in Y}, Q)
       from LUBFraenkelDistr;
  end;
```

This proof is about a more abstract kind of mathematics than the example

from the previous chapter. Mizar is especially good at formalizing abstract mathematics.

2.1 Set comprehension: the Fränkel operator

The proof of theorem `Th5` from `quantal1.miz` contains expressions like:

```
{ H(a,b) where b is Element of Q: b in Y }
```

This corresponds to:

$$\{H(a,b) \mid b \in Y\}$$

where a is a fixed parameter and b is allowed to range over the elements of Q that satisfy $b \in Y$.

In Mizar this style of set comprehension is called the *Fränkel operator*. It corresponds to the axiom of replacement in set theory, which was first proposed by Adolf Fränkel, the F in ZF.

The general form of the Fränkel operator is:

`{` *term* `where` *declaration of the variables*`:` *formula* `}`

As you can see in some of the other Fränkel operator terms in the proof, if all variables in the term occur in a `reserve` statement their declarations can be left implicit.

To be allowed to write a Fränkel operator, the types of the variables involved need to *widen* to a Mizar type that has the form `Element of` A. Only then can Mizar know that the defined set is really a set. Because else you could write proper classes like:

```
{ X where X is set: not X in X }
```

But in Mizar this expression is illegal because `set` doesn't widen to a type of the form `Element of`.

Exercise 2.1.1 Write in Mizar syntax the statement that the set of prime numbers is infinite. Prove it.

2.2 Beyond first order: schemes

The proof of theorem `Th5` from `quantal1.miz` uses the `from` justification.

Mizar is based on first order predicate logic. However first order logic is slightly too weak to do ZF-style set theory. With first order logic ZF-style set theory needs infinitely many axioms. That's because ZF set theory contains an *axiom scheme*, the axiom of replacement.

Here is a table that shows how `scheme` and `from` is similar to `theorem` and `by`:

order	*item*	*used*
first order	`theorem`	`by`
higher order	`scheme`	`from`

Schemes are higher order but only in a very weak sense. It has been said that Mizar schemes are not second order but 1.001th order.

Let's study the first `from` that occurs in the proof. It justifies an equality:

```
{F(c): P[c]} = {G(a): P[a]} from FRAENKEL:sch 5(A1);
```

The justification refers to the statement:

```
A1: for a holds F(a) = G(a);
```

The scheme that is used here is:

```
scheme :: FRAENKEL:sch 5
 FraenkelF' { B() -> non empty set,
   F(set) -> set, G(set) -> set, P[set] } :
  { F(v1) where v1 is Element of B() : P[v1] }
  = { G(v2) where v2 is Element of B() : P[v2] }
 provided
  for v being Element of B() holds F(v) = G(v);
```

The statement that this scheme proves has been underlined. It clearly is the same statement as the one that is being justified in the example. The general structure of a scheme definition is:

```
scheme label { parameters } :
 statement
provided
 statements
proof
 ...
end;
```

In the case of the `FRAENKEL:sch 5` scheme, the parameters are `B`, `F`, `G` and `P`. The first three are functions (those are written with round brackets), and the last is a predicate (written with square brackets).

To use a scheme you have to define *macros* with `deffunc` and `defpred`, that match the parameters of the scheme. In macro definitions the arguments to the macro are written as `$1`, `$2`, ... In this specific case the macros that are defined are:

```
deffunc F(Element of Q) = $1[*]"\/"(Y,Q);
deffunc G(Element of Q) = "\/"({$1[*]b: b in Y}, Q);
defpred P[set] means $1 in X;
```

(Apparently `B()` there doesn't need to be a macro. So Mizar can figure out by itself that in this specific example `B()` has to be instantiated with `carrier of Q`.)

For another example of a scheme let's look at the most commonly used scheme, which is `NAT_1:sch 1`. It allows one to do induction over the natural numbers. This scheme is:

```
scheme :: NAT_1:sch 1
 Ind { P[Nat] } :
  for k being Nat holds P[k]
 provided
  P[0]
 and
  for k being Nat st P[k] holds P[k + 1];
```

So let's show how to use this scheme to prove some statement about the natural nummbers. For example consider the non-negativity of the natural numbers:

```
for n being Nat holds n >= 0;
```

Clearly to prove this using the induction scheme, we need to define the following macro:

```
defpred P[Nat] means $1 >= 0;
```

And then, the structure of the proof will be:

```
    defpred P[Nat] means $1 >= 0;
A1: P[0] by ...;
A2: for k being Nat st P[k] holds P[k + 1]
    proof
     let k be Nat;
     assume k >= 0;
     ...
     thus k + 1 >= 0 by ...;
    end;
    for n being Nat holds P[n] from NAT_1:sch 1(A1,A2);
```

In the statement that is justified and the arguments to the scheme, the macros really have to be present (else Mizar won't know how to match things to the statements in the scheme), so you cannot write `0 >= 0` after `A1`, but everywhere else it doesn't matter whether you write `P[k]` or `k >= 0`, because almost immediately the macro `P[k]` will be expanded to its definition.

Exercise 2.2.1 Write the natural deduction rules as Mizar schemes and prove them. For instance the rule of implication elimination ('modus ponens') becomes:

```
scheme implication_elimination { P[], Q[] } : Q[]
provided
A1: P[] implies Q[] and
A2: P[]
 by A1,A2;
```

Exercise 2.2.2 The axioms of the Mizar set theory are in the article called TARSKI. It's called that way because it contains the axioms of a theory called Tarski-Grothendieck set theory, which is ZF set theory with an axiom about the existence of big cardinal numbers. The TARSKI article contains one scheme:

```
scheme :: TARSKI:sch 1
 Fraenkel { A()-> set, P[set, set] }:
  ex X st for x holds x in X iff ex y st y in A() & P[y,x]
 provided
  for x,y,z st P[x,y] & P[x,z] holds y = z;
```

Is it possible to prove this scheme using the Fränkel operator from the previous section (and without using other schemes from the MML)?

Conversely, is it possible to eliminate the use of the Fränkel operator from Mizar proofs by using the TARSKI:sch 1 scheme?

2.3 Structures

The proof of theorem Th5 from quantal1.miz uses the exciting type:

```
left-distributive right-distributive complete Lattice-like
    (non empty QuantaleStr)
```

In this type left-distributive, right-distributive, complete, Lattice-like and empty are just attributes like before, but the mode QuantaleStr is something new.

The mode QuantaleStr is a *structure*. Structures are the records of Mizar. They contain *fields* that you can select using the construction:

the *field* of *structure*

An example is the expression:

```
the carrier of Q
```

In this the structure is Q and the field name is carrier. Many Mizar structures have the carrier field.

The definition of the QuantaleStr structure is:

```
struct (LattStr, HGrStr) QuantaleStr
   (# carrier -> set,
     L_join, L_meet, mult -> BinOp of the carrier #);
```

Apparently it has four fields: `carrier`, `L_join`, `L_meet` and `mult`.

The modes `LattStr` and `HGrStr` are called the *ancestors* of the structure. They are structures that have a subset of the fields of the structure that is defined. Any term that has type `QuantaleStr` will automatically also get the types `LattStr` and `HGrStr`.

The definitions of the ancestors of `QuantaleStr` are:

```
struct(/\-SemiLattStr,\/-SemiLattStr) LattStr
  (# carrier -> set, L_join, L_meet -> BinOp of the carrier #);
```

and:

```
struct(1-sorted) HGrStr (# carrier -> set,
                           mult -> BinOp of the carrier #);
```

Structures are a powerful concept. They give the MML its abstract flavor.

Exercise 2.3.1 Find the definition of the mode `Field` in `VECTSP_1`. What is the structure it is based on? Recursively trace all ancestors of that structure. Draw a graph of the widening relation between those ancestors.

Exercise 2.3.2 The most basic structure in the MML is from `STRUCT_0`:

```
definition
 struct 1-sorted (# carrier -> set #);
end;
```

Is an object of type `1-sorted` uniquely determined by its `carrier`? Why or why not?

Exercise 2.3.3 You can construct an object of type `1-sorted` as:

```
1-sorted (# A #)
```

Use this notation to define functors `in` and `out` that map objects of type `set` to their natural counterparts in `1-sorted` and back. Prove in Mizar that `out` is a left inverse to `in`.

Exercise 2.3.4 Are structures ZF sets or not? More specifically: if `Q` is a structure, can there be an `x` with `x in Q`?

If not, why not?

If so, can you prove the following?

```
ex Q being 1-sorted, x being set st x in Q
```

If a structure has elements: is it uniquely determined by its elements?

Exercise 2.3.5 Because of Gödel's theorems we know that Mizar's set theory is incomplete. That is, there has to be a Mizar formula ϕ without free variables, for which no Mizar proof exists of ϕ, and also no Mizar proof exists of `not` ϕ.

Can you think of a non-Gödelian ϕ that is provable nor disprovable in Mizar?

The QED Manifesto*

May 15, 1994

The development of mathematics toward greater precision has led, as is well known, to the formalization of large tracts of it, so that one can prove any theorem using nothing but a few mechanical rules.

– K. Gödel

If civilization continues to advance, in the next two thousand years the overwhelming novelty in human thought will be the dominance of mathematical understanding.

– A. N. Whitehead

1 What Is the QED Project and Why Is It Important?

QED is the very tentative title of a project to build a computer system that effectively represents all important mathematical knowledge and techniques. The QED system will conform to the highest standards of mathematical rigor, including the use of strict formality in the internal representation of knowledge and the use of mechanical methods to check proofs of the correctness of all entries in the system.

The QED project will be a major scientific undertaking requiring the cooperation and effort of hundreds of deep mathematical minds, considerable ingenuity by many computer scientists, and broad support and leadership from research agencies. In the interest of enlisting a wide community of collaborators and supporters, we now offer reasons that the QED project should be undertaken.

First, the increase of mathematical knowledge during the last two hundred years has made the knowledge, let alone understanding, of all, or even of the most important, mathematical results something beyond the capacity of any human. For example, few mathematicians, if any, will ever understand the entirety of the recently settled structure of simple finite groups or the proof of the four color theorem. Remarkably, however, the creation of mathematical logic and the advance of computing technology have also provided the means for building a computing system that represents all important mathematical knowledge in an entirely rigorous and mechanically usable fashion. The QED system we imagine will provide a means by which mathematicians and scientists can scan the entirety of mathematical knowledge for relevant results and, using tools of the QED system, build upon such results with reliability and confidence but without the need for minute comprehension of the details or even the ultimate foundations of the parts of the system upon which they build. Note that the approach will almost surely be an incremental one: the most important and applicable results will likely become available before the more obscure and purely theoretical ones are tackled, thus leading to a useful system in the relatively near term.

Second, the development of high technology is an endeavor of fabulously increasing mathematical complexity. The internal documentation of the next generation of microprocessor chips may run, we have heard, to thousands of pages. The specification of a major new industrial system, such as a fly-by-wire airliner or an autonomous undersea mining operation, is likely to be even an order of magnitude greater in complexity, not the least reason being that such a system would perhaps include dozens of microprocessors. We believe that an industrial designer will be able to take parts of the QED system and use them to build reliable formal mathematical models of not only a new industrial system but even the interaction of that system with a formalization of the external world. We believe that such large mathematical models will provide a key principle for the construction of systems substantially more

*A version of this appeared as 'The QED Manifesto' in *Automated Deduction – CADE 12,* Springer-Verlag, Lecture Notes in Artificial Intelligence, Vol. 814, pp. 238–251, 1994.

complex than those of today, with no loss but rather an increase in reliability. As such models become increasingly complex, it will be a major benefit to have them available in stable, rigorous, public form for use by many. The QED system will be a key component of systems for verifying and even synthesizing computing systems, both hardware and software.

The third motivation for the QED project is education. Nothing is more important than mathematics education to the creation of infrastructure for technology-based economic growth. The development of mathematical ability is notoriously dependent upon 'doing' rather than upon 'being told' or 'remembering'. The QED system will provide, via such techniques as interactive proof checking algorithms and an endless variety of mathematical results at all levels, an opportunity for the one-on-one presenting, checking, and debugging of mathematical technique, which it is so expensive to provide by the method of one trained mathematician in dialogue with one student. QED can provide an engaging and non-threatening framework for the carrying out of proofs by students, in the same spirit as a long-standing program of Suppes at Stanford for example. Students will be able to get a deeper understanding of mathematics by seeing better the role that lemmas play in proofs and by seeing which kinds of manipulations are valid in which kinds of structures. Today few students get a grasp of mathematics at a detailed level, but via experimentation with a computerized laboratory, that number will increase. In fact, students can be used (eagerly, we think) to contribute to the development of the body of definitions and proved theorems in QED. Let also us make the observation that the relationship of QED to education may be seen in the following broad context: with increasing technology available, governments will look not only to cut costs of education but will increasingly turn to make education and its delivery more cost-effective and beneficial for the state and the individual.

Fourth, although it is not a practical motivation, nevertheless perhaps the foremost motivation for the QED project is cultural. Mathematics is arguably the foremost creation of the human mind. The QED system will be an object of significant cultural character, demonstrably and physically expressing the staggering depth and power of mathematics. Like the great pyramids, the effort required (especially early on) may be great; but the rewards can be even more staggering than this effort. Mathematics is one of the most basic things that unites all people, and helps illuminate some of the most fundamental truths of nature, even of being itself. In the last one hundred years, many traditional cultural values of our civilization have taken a severe beating, and the advance of science has received no small blame for this beating. The QED system will provide a beautiful and compelling monument to the fundamental reality of truth. It will thus provide some antidote to the degenerative effects of cultural relativism and nihilism. In providing motivations for things, one runs the danger of an infinite regression. In the end, we take some things as inherently valuable in themselves. We believe that the construction, use, and even contemplation of the QED system will be one of these, over and above the practical values of such a system. In support of this line of thought, let us cite Aristotle, the Philosopher, the Father of Logic, 'That which is proper to each thing is by nature best and most pleasant for each thing; for man, therefore, the life according to reason is best and pleasantest, since reason more than anything else is man.' We speculate that this cultural motivation may be the foremost motivation for the QED project. Sheer aesthetic beauty is a major, perhaps the major, force in the motivation of mathematicians, so it may be that such a cultural, aesthetic motivation will be the key motivation inciting mathematicians to participate.

Fifth, the QED system may help preserve mathematics from corruption. We must remember that mathematics essentially disappeared from Western civilization once, during the dark ages. Could it happen again? We must also remember how unprecedented in the history of mathematics is the clarity, even perfection, that developed in this century in regard to the idea of formal proof, and the foundation of essentially the entirety of known mathematics upon set theory. One can easily imagine corrupting forces that could undermine these achievements. For example, one might suspect that there is already a trend towards believing some recent 'theorems' in physics because they offer some predictive power rather than that they have any meaning, much less rigorous proof, with a possible erosion in established standards of rigor. The QED system could offer an antidote to any such tendency. The standard, impartial answer to the question 'Has it been proved?' could become 'Has it been checked by the QED system?' Such a mechanical proof checker could provide answers immune to pressures of emotion, fashion, and politics.

Sixth, the 'noise level' of published mathematics is too high. It has been estimated that something between 50 and 100 thousand mathematical papers are published per year. Nobody knows for sure how many contain errors or how many are repetitions, but some pessimists claim the number of both is high. QED can help to reduce the level of noise, both by helping to find errors and by helping to support computer searches for duplication.

Seventh, QED can help to make mathematics more coherent. There are similar techniques used in various fields of mathematics, a fact that category theory has exploited very well. It is quite natural for formalizers to generalize definitions and propositions because it can make their work much easier.

Eighth, by its insistence upon formalization, the QED project will add to the body of explicitly formulated mathematics. There is mathematical knowledge that is neither taught in classes nor published in monographs. It is below what mathematicians call 'folklore,' which is explicitly formulated. Let us call this lower level of unformulated knowledge 'mathlore'. In formalization efforts, we must formalize everything, and that includes mathlore lemmas.

Ninth, the QED project will help improve the low level of self-consciousness in mathematics. Good mathematicians understand trends and connections in their field. The QED project will enable mathematicians to analyze, perhaps statistically, the whole structure of the mathematics, to discover new trends, to forecast developments and so on.

2 Some Objections to the Idea of the QED Project and Some Responses

> *The peculiarity of the evidence of mathematical truths is that all the argument is on one side. There are no objections, and no answer to objections.*
>
> – J. S. Mill

Objection 1: *Paradoxes, Incompatible Logics, etc. Anyone familiar with the variety of mathematical paradoxes, controversies, and incompatible logics of the last hundred years will realize that it is a myth that there is certainty in mathematics. There is no fundamentally justifiable view of mathematics which has wide support, and no widely agreeable logic upon which such an edifice as QED could be founded.*

First Reply to Objection 1: Although there are a variety of logics, there is little doubt that one can describe all important logics within an elementary logic, such as primitive recursive arithmetic, about which there is no doubt, and within which one can reliably check proofs presented in the more controversial logics. We plan to build the QED system upon such a 'root logic', as we discuss below extensively. But the QED system is to be fundamentally unbiased as to the logics used in proofs. Or if there is to be a bias, it is to be a bias towards universal agreement. Proofs in all varieties of classical, constructive, and intuitionist logic will be found rigorously presented in the QED system – with sharing of proofs between logics where justified by metatheorems. For example, Goedel showed how to map theorems in classical number theory into intuitionist number theory, and E. Bishop showed how to develop much of modern mathematics in a way that is simultaneously constructive and classical. A mathematical logic may be regarded as being very much like a model of the world – one can often profit from using a model even if one ultimately chooses an alternative model because it is more suited to one's purposes. Furthermore, merely because some logic is so overly strong as to be ultimately found inconsistent or so weak as to ultimately fail to be able to express all that one hopes, one can nevertheless often transfer almost all of the technique developed in one logic to a subsequent, better logic.

Second Reply to Objection 1. These are controversies in the Philosophy of Mathematics. Who cares? The overwhelming majority of contemporary mathematicians believe that there are no doubts about what it means for a proof to be correct, and they agree on a vast common mathematical basis, much stronger than ZFC. If we do not get the mathematicians involved, the QED project will fail as well. But to get mathematicians involved, we have to find out how to talk to them.

Objection 2. *Intellectual property problems. Such an enterprise as QED is doomed because as soon as it is even slightly successful, it will be so swamped by lawyers with issues of ownership, copyright, trade secrecy, and patent law that the necessary wide cooperation of hundreds of mathematicians, computer scientists, research agencies, and institutions will become impossible.*

Reply to Objection 2. In full cognizance of the dangers of this objection, we put forward as a fundamental and initial principle that the entirety of the QED system is to be in the international public domain, so that all can freely benefit from it, and thus be inspired to contribute to its further development.

Objection 3. *Too much mathematics. Mathematics is now so large that the hope of incorporating all of mathematics into a system is utterly humanly impossible, especially since new mathematics is generated faster than it can be entered into any system.*

Reply to Objection 3. While it is certainly the case that we imagine anyone being free to add, in a mechanically checked, rigorous fashion, any sort of new mathematics to the QED system, it seems that as a first good objective, we should pursue checking 'named' theorems and algorithms, the sort of things that are commonly taught in universities, or cited as important in current mathematics and applications of mathematics.

Objection 4. *Mechanically checked formality is impossible. There is no evidence that extremely hard proofs can be put into formal form in less than some utterly ridiculous amount of work.*

Reply to Objection 4. Based upon discussions with numerous workers in automated reasoning, it is our view that using current proof-checking technology, we can, using a variety of systems and expert users of those systems, check mathematics at within a factor of ten, often much better, of the time it takes a skilled mathematician to write down a proof at the level of an advanced undergraduate textbook. QED will support proof checking at the speeds and efficiencies of contemporary proof-checking systems. In fact, we see one of the benefits of the QED project as being a demonstration of the viability of mechanically-assisted (-enforced) proof-checking.

Objection 5. *If QED were feasible, it would have already been underway several decades ago.*

Reply to Objection 5. Many of the most well-known projects related to QED were commenced in an era in which computing was exorbitantly expensive and computer communication between geographically remote groups was not possible. Now most secretaries have more computing power than was available to most entire QED-related projects at their inception, and rapid communication between most mathematics and computer science departments through email, telnet, and ftp has become almost universal. It also now seems unlikely that any one small research group can, alone, make a major dent in the goal of incorporating all of mathematics into a single system, but at the same time technology has made widespread collaboration entirely feasible, and the time seems ripe for a larger scale, collaborative effort. It is also worth adding that research agencies may now be in a better position to recognize the Babel of incompatible reasoning systems and symbolic computation systems that have evolved from a plethora of small projects without much attention to collaboration. Then perhaps they can work towards encouraging collaboration, to minimize the lack of interoperability due to diversity of theorem-statement languages, proof languages, programming languages, computing platforms, quality, and so on.

Objection 6. *QED is too expensive.*

Reply to Objection 6. While this objection requires careful study at some point, we note that simply concentrating the efforts of some currently-funded projects could go a long way towards getting QED off the ground. Moreover, as noted above, students could contribute to the project as an integrated part of their studies once the framework is established, presumably at little or no cost. We can imagine a number of professionals contributing as well. In particular, there is currently a large body of tenured or retired mathematicians who have little inclination for advanced research, and we believe that some of these could be inspired to contribute to this project. It may be a good idea to have a QED governing board to recognize contributions.

Objection 7. *Good mathematicians will never agree to work with formal systems because they are syntactically so constricting as to be inconsistent with creativity.*

Reply to Objection 7. The written body of formal logic rightly repulses most mathematical readers. Whitehead and Russell's Principia Mathematica did not establish mathematics in a notation that others happily adopted. The traditional definition of formal logics is in a form that no one can stand to use in practice, e.g., with function symbols named f_1, f_2, f_3, ... The absence of definitional principles for almost all formal logics is an indication that from the beginning, formal logics became something to be studied (for properties such as completeness) rather than to be

used by humans, the practical visions of Leibniz and Frege notwithstanding. The developers of proof checking and theorem-proving systems have done little towards making their syntax tolerable to mathematicians. Yet, on this matter of syntax, there is room for the greatest hope. Although the subject of mechanical theorem-proving in general is beset with intractable or unsolvable problems, a vastly improved computer-human interface for mathematics is something easily within the grasp of current computer theory and technology. The work of Knuth on TEX and the widespread adoption of TEX by mathematicians and mathematics journals demonstrates that it is no problem for computers to deal with any known mathematical notation. Certainly, there is hard work to be done on this problem, but it is also certainly within the capacity of computer science to arrange for any rigorously definable syntax to be something that can be conveniently entered into computers, translated automatically into a suitable internal notation for formal purposes, and later reproduced in a form pleasant to humans. It is certainly feasible to arrange for the users of the QED system to be able to shift their syntax as often as they please to any new syntax, provided only that it is clear and unambiguous. Perhaps the major obstacle here is simply the current scientific reward system: precisely because new syntaxes, new parsers, and new formatters are so easy to design, little or no credit (research, academic, or financial) is currently available for working on this topic. Let us add that we need take no position on the question whether mathematicians can or should profit from the use of formal notations in the discovery of serious, deep mathematics. The QED system will be mainly useful in the final stages of proof reporting, similar to writing proofs up in journals, and perhaps possibly never in the discovery of new insights associated with deep results.

Objection 8. *The QED system will be so large that it is inevitable that there will be mistakes in its structure, and the QED system will, therefore, be unreliable.*

Reply to Objection 8. There is no doubt considerable room for error in the construction of the QED system, as in any human enterprise. A key motivation in Babbage's development of the computer was his objective of producing mathematical tables that had fewer errors than those produced by hand methods, an objective that has certainly been achieved. It is our experience that even with the primitive proof checking systems of today, errors made by humans are frequently found by the use of such tools, errors that would perhaps not otherwise be caught. The standard of success or failure of the QED project will not be whether it helps us to reach the kingdom of perfection, an unobtainable goal, but whether it permits us to construct proofs substantially more accurately than we can with current hand methods. In defense of the QED vision, let us assert that we believe that room for error can be radically reduced by (a) expressing the full foundation of the QED system in a few pages of mathematics and (b) supporting the development of essentially independent implementations for the basic checker. It goes without saying that in the development of any particular subfield of mathematics, errors in the statements of definitions and other axioms are possible. Agreement by experts in each mathematical subfield that the definitions are 'right' will be a necessary part of establishing confidence that mechanically checked theorems establish what is intended. There is no mechanical method for guaranteeing that a logical formula says what a user intuitively means.

Objection 9. *The cooperation of mathematicians is essential to building the QED edifice of proofs. However, because it is likely to remain very tedious to prove theorems formally with mechanical proof checkers for the foreeable future, mathematicians will have no incentive to help.*

Reply to Objection 9. To be developed, QED does not need to attract the support of all or most mathematicians. If only a tenth of one percent of mathematicians could be attracted, that will probably be sufficient. And in compensation for the extra work currently associated with entering formal mathematics in proof checking systems, we can point out that some mathematicians may find the following benefit sufficiently compensatory: in formally expressing mathematics, one's own thoughts are often sharply clarified. One often achieves an appreciation for subtle points in proofs that one might otherwise skim over or skip. And the sheer joy of getting all the details of a hard theorem 'exactly right', because formalized and machine checked, is great for many individuals. So we conjecture that enough mathematicians will be attracted to the endeavor provided it can be sufficiently organized to have a real chance of success.

Objection 10. *The QED project represents an unreasonable diversion of resources to the*

pursuit of the checking of ordinary mathematics when there is so much profitably to be done in support of the verification of hardware and software.

Reply to Objection 10. Current efforts in formal, mechanical hardware and software verification are exceptionally introspective, focusing upon internal matters such as compilers, operating systems, networks, multipliers, and busses. From a mathematical point of view, essentially all these verifications fall into a tiny, minor corner of elementary number theory. But eventually, verification must reach out to consider the intended effect of computing systems upon the external, continuous world with which they interact. If one attempts to try to verify the use of a DSP chip for such potentially safety critical applications as telecommunications, robot vision, speech synthesis, or cat scanning, one immediately sees the need for such basic engineering mathematics as Fourier transforms, not something at which existing verification systems are yet much good. By including the rigorous development of the mathematics used in engineering, the QED project will make a crucial contribution to the advance of the verification of computing systems.

Objection 11. *The notion that interesting mathematics can ever, in practice, be formally checked is a fantasy. Whitehead and Russell spent hundreds of pages to prove something as trivial as that* 0 *is not* 1*. The notion that computing systems can be verified is another fantasy, based upon the misconception that mathematical proof can guarantee properties of physical devices.*

Reply to Objection 11. That many interesting, well-known results in mathematics can be checked by machine is manifest to those who take the trouble to read the literature. One can mention merely as examples of mathematics mechanically checked from first principles: Landau's book on the foundations of analysis, Girard's paradox, Rolle's theorem, both Banach's and Knaster's fixed point theorems, the mean value theorem for derivatives and integrals over Banach-space valued functions, the fundamental counting theorem for groups, the Schroeder-Bernstein theorem, the Picard-Lindelof theorem for the existence of ODEs, Wilson's theorem, Fermat's little theorem, the law of quadratic reciprocity, Ramsey's theorem, Goedel's incompleteness theorem, and the Church-Rosser theorem. That it is possible to verify mechanically a simple, general purpose microprocessor from the level of gates and registers up through an application, via a verified compiler, has been demonstrated. So there is no argument against proof-checking or mechanical verification in principle, only an ongoing and important engineering debate about cost-effectiveness. The noisy verification debate is largely a comedy of misunderstanding. In reaction to a perceived sanctimony of some verification enthusiasts, some opponents impute to all enthusiasts grandiose claims that complete satisfaction with a computing product can be established by mathematical means. But any verification enthusiast ought to admit that, at best, verification establishes a consistency between one mathematical theory and another, e.g., between a formal specification of intended behavior of a system and a formal representation of an implementation, say in terms of gates and memory. Mathematical proof can establish neither that a specification is what any user 'really wants' nor that a description of gates and memory corresponds to physical reality. So whether the results of a computation will be pleasing to or good for humans is something that cannot be formally stated, much less proved.

Objection 12. *The QED Manifesto is too long. Its length will interfere with the establishment of the project by driving away potential supporters and contributors.*

Reply to objection 12. Objection 12 is largely correct. For an initial reading, it is suggested that sections 4 and 5 below be skipped. On the other hand, we believe that there is real value in recording the many views on this subject, even views that are clearly refutable.

3 Some Background, Being a Critique of Current Related Efforts

> *Although the root of logic is the same for all, the 'hoi polloi' live as though they have a private understanding.*
>
> – Heraclitus

In some sense project QED is already underway, via a very diverse collection of projects. Unfortunately, progress seems greatly slowed by duplication of effort and by incompatibilities. If the many people already involved in work related to QED had begun cooperation twenty-five years ago in pursuing the construc-

tion of a single system (or federation of subsystems) incorporating the work of hundreds of scientists, a substantial part of the system, including at least all of undergraduate and much of first year graduate mathematics and computer science, could already have been incorporated into the QED system by now. We offer as evidence the nontrivial fragments of that body of theorems that has been successfully completed by existing proof-checking systems.

The idea of QED is perhaps 300 years old, but one can imagine tracing it back even 2500 years. We can agree that many groups and individuals have made substantial progress on parts of this project, yet we can ask the question, is there today any project underway which can be reasonably expected to serve as the basis for QED? We believe not, we are afraid not, though we would be delighted to join any such project already underway. One of the reasons that we do not believe there is any such project underway is that we think that there exist a few basic, unsolved technical problems, which we discuss below. A second reason is that few researchers are interested in doing the hard work of checking proofs – probably due to an absence of belief that much of the entire QED edifice will ever be constructed. Another reason is that we are familiar with many automated reasoning projects but see very serious problems in many of them. Here are some of these problems.

1. *Too much code to be trusted.* There have been a number of automated reasoning systems that have checked many theorems of interest, but the amount of code in some of these impressive systems that must be correct if we are to have confidence in the proofs produced by these systems is vastly greater than the few pages of text that we wish to have as the foundation of QED.

2. *Too strong a logic.* There have been many good automated reasoning systems that 'wired in' such powerful rules of inference or such powerful axioms that their work is suspect to many of those who might be tempted to contribute to QED – those of an intuitionistic or constructivist bent.

3. *Too limited a logic.* Some projects have been developed upon intuitionistic or constructive lines, but seem unlikely, so far anyway, to support also the effective checking of theorems in classical mathematics. We regard this 'boot-strapping problem' – how to get, rigorously, from checking theorems in a weak logic to theorems in a powerful classical logic, in an effective way – to be a key unsolved technical obstacle to QED. We discuss it further below.

4. *Too unintelligible a logic.* Some people have attempted to start projects on a basis that is extremely obscure, at least when observed by most of the community. We believe that if the initial, base, root logic is not widely known, understood, and accepted, there will never be much enthusiasm for QED, and hence it will never get off the ground. It will take the cooperation of many, many people to build the QED system.

5. *Too unnatural a syntax.* Just as QED must support a variety of logics, so too must it support a variety of syntaxes, enough to make most groups of mathematicians happy when they read theorems they are looking for. It is unreasonable to expect mathematicians to have to use some computer oriented or otherwise extremely simplified syntax when concentrating on deep mathematical thoughts. Of course, a rigorous development of the syntaxes will be essential, and it will be a burden on human readers using the QED proof tree to 'know' not only the logical theory in which any theorem or procedure they are reading is written but also to know the syntax being used.

6. *Parochialism.* There are many projects that have started over from scratch rather than building upon the work of others, for reasons of remoteness, ignorance of previous work, personalities, unavailability of code due to intellectual property problems, and issues of grants and publications. We are extremely sensitive to the fact that the issue of credit for scientific work in a large scale project such as this can be a main reason for the failure of the QED project. But we can be hopeful that if a sufficient number of scientists unite in supporting the QED project, then partial contributions to QED's advancement will be seen in a very positive light in comparison to efforts to start all over from scratch.

7. *Too little extensibility.* In 20 years there have been perhaps a dozen major proof-checking projects, each representing an

enormous amount of activity, but which have 'plateaued out' or even evaporated. It seems that when the original authors of these systems cease actively working on their systems, the systems tend to die. Perhaps this problem stems from the fact that insufficient analysis was given to the basic problems of the root logic. Without a sufficient amount of extensibility, everyone so far seems to have reached a point in which checking new proofs is too much work to do by machine, even though one knows that it is relatively easy for mathematicians to keep making progress by hand. The reason, we suspect, is that mathematicians are using some reflection principles or layers of logics in ways not yet fully understood, or at least not implemented. Mathematicians great contribution has been the continual re-evaluating, re-conceptualizing, connecting, extending and, in cases, discarding of theorems and areas. So each generation stands on the shoulders of the giants before, as if they had always been there. We are far from being able to represent mechanically such evolutionary mathematical processes. Existing mathematical logics are typically as 'static' as possible, often not even permitting the addition of new definitions! Important work in logic needs to be done to design logics more adaptable to extension and evolution.

8. *Too little heuristic search support.* While it is in principle possible to generate entries in the QED system entirely by hand, it seems extremely likely that some sort of automated tools will be necessary, including tools that do lots of search and use lots of heuristics or strategies to control search. Some systems which have completely eschewed such search and heuristic techniques might have gotten much further in checking interesting theorems through such techniques.

9. *Too little care for rigor.* It is notoriously easy to find 'bugs' in algorithms for symbolic computation. To make matters worse, these errors are often regarded as of no significance by their authors, who plead that the result returned is true 'except on a set of measure zero', without explicitly naming the set involved. The careful determination, nay, even proof, of precisely which conditions under which a result is true is essential for building the structure of mathematics so that one can depend on it. The QED system will support the development of symbolic algebra programs in which formal proofs of correctness of derivations are provided, along with the precise statement of conditions under which the results are true.

10. *Complete absence of inter-operability.* One safe generalization about current automated reasoning or symbolic computation systems is that it is always somewhere between impossible and extremely difficult to use any two of them together reliably and mechanically. It seems almost essential to the inception of any major project in this area to choose a logic and a syntax that is original, i.e., incompatible with other tools. One major exception to this generalization is the base syntax and logic for resolution systems. Here, standard problem sets have been circulated for years. But even for such resolution systems there is no standard syntax for entering problems involving such fundamental mathematical constructs as induction schemas or set-builder notation.

11. *Too little attention paid to ease of use.* The ease of use of automated reasoning systems is perhaps lower than for any other type of computing system available! In general, while anyone can use a word processor, almost no one but an expert can use a proof checker to check a difficult theorem. Perhaps this can be explained by the fact that the designers of such systems have had to put so much of their energies and attention into rigor, that they simply did not have enough energy left for good interface design.

4 The Relationship of QED to Artificial Intelligence (AI) and to Automated Reasoning (AR)

Project QED is largely independent of the question of the possibility or utility of artificial intelligence or automated reasoning. To the extent that mechanical aids of any kind can be used to help construct (or shorten) entries in the QED system, we can be appreciative of such aids, even if the aids use techniques

that are from the realms of artificial intelligence, assuming of course that what the aids suggest doing is verifiably correct. A key fact is that it will not matter, from the viewpoint of soundness, whether proofs were added to the QED system by humans, dumb programs, smart programs or some combination thereof. All of the QED system will be checkable by a simple program, from first principles. The QED system will focus on what is known in mathematics, both theorems and techniques, rather than upon the problems of discovering new mathematics.

It is the view of some of us that many people who could have easily contributed to project QED have been distracted away by the enticing lure of AI or AR. It can be agreed that the grand visions of AI or AR are much more interesting than a completed QED system while still believing that there is great aesthetic, philosophical, scientific, educational, and technological value in the construction of the QED system, regardless of whether its construction is or is not largely done 'by hand' or largely automatically.

5 The Root Logic – Some Technical Details

> *Method consists entirely in the order and disposition of the objects towards which our mental vision must be directed if we would find out any truth. We shall comply with it exactly if we reduce involved and obscure propositions step by step to those that are simpler, and then starting with the intuitive apprehension of all those that are absolutely simple, attempt to ascend to the knowledge of all others by precisely similar steps.*
>
> – R. Descartes

An important early technical step will be to 'get off the ground', logically speaking, which we will do by rooting the QED system in a 'root logic', whose description requires only a few pages of typical logico-mathematical text. As a model for brevity and clarity, we can refer the reader to Goedel's presentation, in about two pages, of high-order logic with number theory and set theory, at the beginning of his famous paper on undecidable questions.

The reason that we emphasize succinctness in the description of the logic is that we hope that there will be many separate implementations of a proof checker for this 'root logic' and that each of these implementations can check the correctness of the entire QED system. In the end, it will be the 'social process' of mathematical agreement that will lead to confidence in the implementations of these proof-checkers for the root logic of the QED system, and multiple implementations of a succinct logic will greatly increase the chance this social process will occur.

It is crucial that a 'root logic' be a logic that is agreeable to all practicing mathematicians. The logic will, by necessity, be sufficiently strong to check any explicit computation, but the logic surely must not prejudge any historically debated questions such as the law of the excluded middle or the existence of uncountable sets.

As just one hint of a logic that might be used as the basis of QED, we mention Primitive Recursive Arithmetic (PRA) which is the logic Skolem invented for the foundations of arithmetic, which was later adopted by Hilbert-Bernays as the right vehicle for proof theory. It has also been further developed by Goodstein. In PRA one finds (a) an absence of explicit quantification, (b) an ability to define primitive recursive functions, (c) a few rules for handling equality, e.g., substitution of equals for equals, (d) a rule of instantiation, and (e) a simple induction principle. One reason for taking such a logic as the root logic is that it is doubtful that Metamathematics can be developed in a weaker logic. In any root logic one needs to be able to define, inductively, an infinite collection of terms and, inductively, an infinite collection of theorems, using in the definition of 'theorem' such primitive recursive concepts as substitution. Thus PRA has the bare minimum power we would need to 'get off the ground'. Yet we think it suffices even for checking theorems in classical set theory, in a sense we describe below. The logic FS0, conservative over PRA, but with sets and quantifiers, has been proposed by Feferman as a vehicle more congenial than PRA for studying logics.

It is probably the case that the syntax of resolution theorem-proving is the most widely used and most easily understood logic in the history of work on mechanical theorem-proving and proof checking, and thus perhaps a resolution-like logic could serve as a natural choice for a root logic. Some may object on the grounds that resolution, being based upon classical first order logic, 'wires in' the law of the excluded middle, and therefore is objectionable to constructivists. In response to this objection, let us note that constructivists do

not object to the law of the excluded middle in a free variable setting if all of the predicates and function symbols 'in sight' are recursively defined; for example, it is a constructive theorem that for all positive integers x and y, x divides y or x does not divide y. Thus we might imagine taking as a root logic resolution restricted to axioms describing recursive functions and hereditarily finite objects, such as the integers.

The lambda-calculus-based 'logical frameworks' work in Europe, in the de Bruijn tradition, is perhaps the most well developed potential root logic, with several substantial computer implementations which have already checked significant parts of mathematics. And already, many different logics have been represented in these logical frameworks. As a caution, we note that some may worry there is dangerously too much logical power in some of these versions of the typed lambda calculus. But such logical frameworks give rise to the hope that the root logic might be such that classical logic could simply be viewed as the extension of the root logic by a few higher-order axioms such as $\forall P\,(P \vee \neg P)$.

One possible argument in favor of adopting a root logic of power PRA is that its inductive power permits the proof of metatheorems, which will enable the QED system to check and then effectively use decision procedures. For example, the deduction theorem for first order logic is a theorem of FS0, something not provable in some logical framework systems, for want of induction.

Regardless of the strength or weakness of the root logic chosen, we believe that we can rigorously incorporate into the QED system any part of mathematics, including extremely non-constructive set theoretic arguments, because we can represent these arguments 'one level removed' as 'theorems' that a certain finite object is indeed a proof in a certain theory. For example, if we have in mind some high powered theorem, say, the independence of the continuum hypothesis, we can immediately think of a corresponding theorem of primitive recursive arithmetic that says, roughly, that some sequence of formulas is a proof in some suitable set theory, S1, of another theorem about some other set theory, where a, say, primitive recursive proof checker for S1 has been written in the root logic of QED. In practice, it will be highly advantageous if we make it appear that one isn't really proving a theorem of proof theory but rather is proving a theorem of group theory or topology or whatever.

Although many groups have built remarkable theorem-proving and proof checking systems, we believe that there is a need for some further scientific or computational advances to overcome some 'resource' problems in building a system that can hold all important mathematics. Simply stated, it appears that complete proofs of certain theorems that involve a lot of computation will require more disk space for their storage than could reasonably be expected to be available for the project. The most attractive solution to such a problem is the development of 'reflection' techniques that will permit one to use algorithms that have been rigorously incorporated within QED as part of the QED proof system.

Although we have spoken of a single root logic, we need to make clear that we do not want to fall into the trap of searching for a single, ideal logic. We can easily imagine that it will be possible to develop several different root logics each of which can be fully regarded to be 'a' foundation of QED, each of which is capable as acting as a basis for the other, and each of which has very short implementations which have been checked by the 'social process'. And each of which can be used to check the correctness of the entire QED system.

In any case, it is a highly desireable goal that a checker for the root logic can be easily written in common programming languages. The specification should be so unambiguous that many can easily implement it from its specification in a few pages of code, with total comprehension by a single person.

It has been argued that the idea of having multiple logics in addition to the root logic is a mistake that will result in too much complexity, and that it would be far more sensible to have a single logic in which proofs were clearly flagged with an indication of the assumptions used, so that a single logic could be enjoyed by people of both classical and constructive persuasions. Certainly such a single logic is desireable, but whether such a single logic can be developed is a serious question given that some famous constructive theorems (such as the continuity of all functions on the reals) are classical falsehoods.

It has been argued that the idea of searching for a single logic or a single computer system is inferior to the idea of developing translation mechanisms that would permit proof checking systems to exchange proofs with one another. If this were feasible, it would certainly permit an alternative, distributed approach to achieving the major QED objectives. However, the history of radical incompatibility

of many proof checking systems does suggest that such translation mechanisms may be difficult to produce.

In seeking a root logic, it is clear that there will be many controversies that will be impossible to resolve to everyone's satisfaction. For example, there seems no hope of satisfying in a single logic those who insist upon a typed syntax and those who loathe typed syntax, preferring to do typing internally, e.g., with sets. There are also simple questions not yet resolved after centuries of thought, such as the semantics of a function applied outside its domain, e.g., division by zero.

6 What Is To Be Done?

> *The idea is to make a language such that everything we write in it is interpretable as correct mathematics ... This may include the writing of a vast mathematical encyclopedia, to which everybody (either a human or a machine) may contribute what he likes. The idea of a kind of formalized encyclopedia was already conceived and partly carried out by Peano around 1900, but that was still far from what we might call automatically readable.*
>
> – N. G. de Bruijn

Leadership. It seems certain that inviting deliberation by many interested parties at the planning stage is important not only to get the QED project off on a correct footing but also to encourage many to participate in the project. Until we can establish general agreement within a large, critical mass of scientists (including many distinguished mathematicians) that the QED project is probably worth doing, and until a basic 'manifesto' agreeable to them can be drafted, possibly using parts of this document as a starting point, it is not clear whether there will be any further progress on this project. Given the extraordinary scope of this project, it is also essential that research agency leadership be obtained. It is perhaps unlikely that any one agency would be willing to undertake the funding of the entirety of such a large project. So an agreement by many agencies to cooperate will probably be essential. The requirements for leadership, both by scientists and by research agencies, are so major that it is perhaps premature to speculate about what other things should be done, in what order. Nevertheless, we will speculate about a few issues.

What planning steps should be taken to start the QED project? An obvious first concern is to enumerate and describe in some detail the kinds of things that would be found in the QED system, including

- logics
- axioms
- definitions
- theorems (including an analysis of the major parts of mathematics)
- proofs
- proof-checkers
- decision procedures
- theorem-proving programs
- symbolic computation procedures
- modeling software
- simulation software
- tools for experimentation
- numerical analysis software
- graphical tools for viewing mathematics
- interface tools for using the QED system

Crucial to this initial high level organization effort is deciding what parts of mathematics will be represented, how that mathematics will be organized, and how it will be presented. It is conceivable that years of consideration of these points should precede implementation efforts. One can imagine that a re-organization of mathematics on the order of the scope of the Bourbaki project is necessary. One can imagine major projects in the development of formal 'higher-level' languages in which mathematics can be formally discussed and major projects devoted simply to writing the most important theorems, definitions, and proof sketches in such languages. Because different proofs of the same theorem can differ substantially in complexity, and because entering formal proofs into a proof checking system is very expensive, it is highly cost effective to consider many proofs of a theorem before setting out to verify one of them. It has been suggested by several people that a useful and relatively easy early step would be to assemble, in ftp-able form, a comprehensive survey of the parts of mathematics have been checked by various automated reasoning systems.

A second planning step would be to establish some 'milestones' or some priority list of objectives. For example, one could attempt to outline which parts of mathematics should be added to the system in what order. Simultaneously, an analysis of what sorts of cooperation and resources would be necessary to achieve the earlier goals should be performed.

A third planning step would be to accumulate the basic mathematical texts that are to

be formalized. It is entirely possible that the QED project will greatly overlap with an effort to build an electronic library of mathematical information. It is not part of the idea of a library that the documents should be in any particular language or subjected to any sort of rigor check. But it would of great inherent value, and great value to the QED project, to have the important works of mathematics available in machine readable form and organized for ease of access.

A fourth planning step would be to attempt to achieve consensus about the statement of the most important definitions and theorems in mathematics. Until there is agreement on the formalization of the basic concepts and theorems of the important parts of mathematics, it will be hardly appropriate to begin the difficult task of building formal proofs of theorems. The formalization of statements is an extremely difficult and error-prone activity.

Although the scientific obstacles to building QED are formidable, the social, psychological, political, and economic obstacles seem much greater. In principle, we can imagine a vast collection of people successfully collaborating on such an effort. But the problems of actually getting such a collaboration to occur are possibly insurmountable. 'Why,' an individual researcher could well ask, 'should I risk my future by working on what will be but a small part of a vast undertaking? What sort of recognition will I receive for contributing to yet one more computing system?' These are good questions, and it is not clear what the answer is. To a major extent, status in mathematics and computing is a function of publications in major journals – status for research funding, status for tenure decisions, status for promotion. It is far from clear how contributing pieces to the QED system could provide a substitute for such signs of status. Perhaps here research agencies or even university faculties and administrators could be of assistance in establishing a new societal framework in which such cooperation was encouraged.

Even given the cooperation of all the necessary people and assuming good luck in overcoming scientific hurdles, there are many issues of a very difficult but somewhat mundane character involving: version control, distribution, and support. A system with hundreds of contributors will create management difficulties perhaps not even imaginable to the small groups of researchers who have worked in the past on parts of the QED idea.

It has been suggested about the low-level QED data files that they should be humanly readable and permit comments, and that the character set should be email-able.

It has been suggested that the QED system should include historical information. Although such information would obviously not be something that would be mechanically checkable, it could provide extremely valuable contextual information to those trying to learn mathematics from the system, just as the commentaries on Euclid make his Elements intelligible to the modern reader. Strenuous disputes over priority in all forms of discovery, including mathematics, are common, and therefore care must be taken that the QED system permit the presentation of all sides of such disputes.

It has been suggested that it would be best if QED focused initially on one part of mathematics, namely ring theory.

Authorship: This preliminary discussion of project QED (very tentative name) is an amalgam of many ideas that many people have had and for which perhaps no one alive today deserves much credit. We are deliberately avoiding any authorship or institutional affiliation at this early stage in the project (and may decide to do so forever) in the hope that many will want to join in the project as principals, even as originators (to the extent that anyone alive today could be thought to be an originator of this project). Some of those involved in the project would much rather that QED be completed than that they, as individuals, be lucky enough to partake significantly in the project, much less get any public credit for its completion. It may seem paranoid to avoid personalities, but we are inspired by the extraordinary cooperation achieved in the Bourbaki series in an atmosphere of anonymity.

To join an Internet electronic discussion group devoted to the QED project, send a message with the single line

```
subscribe qed
```

to `majordomo@mcs.anl.gov`. The line above should be the content of the message, not the subject line. The subject line is ignored. An archive of this discussion group is in the directory `/pub/qed/archive/` available by anonymous ftp from `info.mcs.anl.gov`.