Yves Bertot
Pierre Castéran

Interactive Theorem Proving and Program Development

Coq'Art: The Calculus of Inductive Constructions

Springer

Yves Bertot · Pierre Castéran

Interactive Theorem Proving and Program Development

Coq'Art: The Calculus of Inductive Constructions

Foreword by
Gérard Huet and
Christine Paulin-Mohring

Springer

Authors

Dr. Yves Bertot
Inria Sophia Antipolis
2004 route des lucioles
06902 Sophia Antipolis Cedex
France
Yves.Bertot@sophia.inria.fr
www-sop.inria.fr/lemme/Yves.Bertot

Dr. Pierre Castéran
LaBRI and Inria Futurs
LabRI
Université Bordeaux I
351 Cours de la Liberation
33405 Talence Cedex
France
Casteran@labri.fr
www.labri.fr/Perso/~casteran

Series Editors

Prof. Dr. Wilfried Brauer
Institut für Informatik der TUM
Boltzmannstr. 3, 85748 Garching, Germany
Brauer@informatik.tu-muenchen.de

Prof. Dr. Grzegorz Rozenberg
Leiden Institute of Advanced Computer Science
University of Leiden
Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
rozenber@liacs.nl

Prof. Dr. Arto Salomaa
Turku Centre for Computer Science
Lemminkäisenkatu 14 A, 20520 Turku, Finland
asalomaa@utu.fi

Library of Congress Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

ACM Computing Classification (1998): D.2.4, F.3.1, F.4.1, I.2.3

ISBN 978-3-642-05880-6 ISBN 978-3-662-07964-5 (eBook)
DOI 10.1007/978-3-662-07964-5

Cover Design: KünkelLopka, Heidelberg
Typesetting: Camera-ready by the authors
Printed on acid-free paper 45/3141/Tr - 5 4 3 2 1 0

Contents

1

A Brief Overview

Coq [37] is a proof assistant with which students, researchers, or engineers can express specifications and develop programs that fulfill these specifications. This tool is well adapted to develop programs for which absolute trust is required: for example, in telecommunication, transportation, energy, banking, etc. In these domains the need for programs that rigorously conform to specifications justifies the effort required to verify these programs formally. We shall see in this book how a *proof assistant* like *Coq* can make this work easier.

The *Coq* system is not only interesting to develop safe programs. It is also a system with which mathematicians can develop proofs in a very expressive logic, often called *higher-order logic.* These proofs are built in an interactive manner with the aid of automatic search tools when possible. The application domains are very numerous, for instance logic, automata theory, computational linguistics and algorithmics (see [1]).

This system can also be used as a logical framework to give the axioms of new logics and to develop proofs in these logics. For instance, it can be used to implement reasoning systems for modal logics, temporal logics, resource-oriented logics, or reasoning systems on imperative programs.

The *Coq* system belongs to a large family of computer-based tools whose purpose is to help in proving theorems, namely *Automath* [34], *Nqthm* [17, 18], *Mizar* [83], *LCF* [48], *Nuprl* [25], *Isabelle* [73], *Lego* [60], *HOL* [47], *PVS* [68], and *ACL2* [55], which are other renowned members of this family. A remarkable characteristic of *Coq* is the possibility to generate certified programs from the proofs and, more recently, certified modules.

In this introductory chapter, we want to present informally the main features of *Coq*. Rigorous definitions and precise notation are given in later chapters and we only use notation taken from usual mathematical practice or from programming languages.

1.1 Expressions, Types, and Functions

The specification language of *Coq*, also called *Gallina*, makes it possible to represent the usual types and programs of programming languages.

Expressions in this language are formed with constants and identifiers, following a few construction rules. Every expression has a type; the type for an identifier is usually given by a *declaration* and the rules that make it possible to form combined expressions come with *typing rules* that express the links between the type of the parts and the type of the whole expression.

For instance, let us consider the type `Z` of integers, corresponding to the set $\mathbb{Z}$. The constant -6 has this type and if one declares a variable z with type `Z`, the expression $-6z$ also has type `Z`. On the other hand, the constant `true` has type `bool` and the expression "`true` $\times -6$" is not a well-formed expression.

We can find a large variety of types in the *Gallina* language: besides `Z` and `bool`, we make intensive use of the type `nat` of natural numbers, considered as the smallest type containing the constant 0 and the values obtained when calling the successor function. Type operators also make it possible to construct the type $A \times B$ of pairs of values (a, b) where a has type A and b has type B, the type "`list` A" of lists where all elements have type A and the type $A \rightarrow B$ of functions mapping any argument of type A to a result of type B.

For instance, the functional that maps any function f from `nat` to `Z` and any natural number n to the value $\Sigma_{i=0}^{i=n} f(i)$ can be defined in *Gallina* and has type `(nat→Z)→nat→Z`.

We must emphasize that we consider the notion of *function* from a computer science point of view: functions are effective computing processes (in other words, *algorithms*) mapping values of type A to values of type B; this point of view differs from the point of view of set theory, where functions are particular subsets of the cartesian product $A \times B$.

In *Coq*, computing a value is done by successive reductions of terms to an irreducible form. A fundamental property of the *Coq* formalism is that computation always terminates (a property known as *strong normalization*). Classical results on computability show that a programming language that makes it possible to describe all computable functions must contain functions whose computations do not terminate. For this reason, there are computable functions that can be described in *Coq* but for which the computation cannot be performed by the reduction mechanism. In spite of this limitation, the typing system of *Coq* is powerful enough to make it possible to describe a large subclass of all computable functions. Imposing strong normalization does not significantly reduce the expressive power.

1.2 Propositions and Proofs

The *Coq* system is not just another programming language. It actually makes it possible to express assertions about the values being manipulated. These values may range over mathematical objects or over programs.

Here are a few examples of assertions or *propositions*:

- $3 \leq 8$,
- $8 \leq 3$,
- "for all $n \geq 2$ the sequence of integers defined by

$$u_0 = n$$
$$u_{i+1} = \begin{cases} 1 & \text{when } u_i = 1 \\ u_i/2 & \text{when } u_i \text{ is even} \\ 3u_i + 1 & \text{otherwise} \end{cases}$$

  ultimately reaches the value 1,"
- "list concatenation is associative,"
- "the algorithm `insertion_sort` is a correct sorting method."

Some of these assertions are true, others are false, and some are still conjectures—the third assertion[1] is one such example. Nevertheless, all these examples are well-formed propositions in the proper context.

To make sure a proposition P is true, a safe approach is to provide a proof. If this proof is complete and readable it can be verified. These requirements are seldom satisfied, even in the scientific literature. Inherent ambiguities in all natural languages make it difficult to verify that a proof is correct. Also, the complete proof of a theorem quickly becomes a huge text and many reasoning steps are often removed to make the text more readable.

A possible solution to this problem is to define a formal language for proofs, built along precise rules taken from proof theory. This makes it possible to ensure that every proof can be verified step by step.

The size of complete proofs makes it necessary to mechanize their verification. To trust such a mechanical verification process, it is enough to show that the verification algorithm actually verifies that all formal rules are correctly applied.

The size of complete proofs also makes it inpractical to write them manually. In this sense, naming a tool like *Coq* a *proof assistant* becomes very meaningful. Given a proposition that one wants to prove, the system proposes tools to construct a proof. These tools, called *tactics*, make it easier to construct the proof of a proposition, using elements taken from a context, namely, declarations, definitions, axioms, hypotheses, lemmas, and theorems that were already proven.

In many cases, tactics make it possible to construct proofs automatically, but this cannot always be the case. Classical results on proof complexity and

[1] The sequence u_i in this assertion is known as the "Syracuse sequence."

computability show that it is impossible to design a general algorithm that can build a proof for every true formula. For this reason, the *Coq* system is an interactive system where the user is given the possibility to decompose a difficult proof in a collection of lemmas, and to choose the tactic that is adapted to a difficult case. There is a wide variety of available tactics and expert users also have the possibility to add their own tactics (see Sect. 7.6).

The user can actually choose not to read proofs, relying on the existence of automatic tools to construct them and a safe mechanism to verify them.

1.3 Propositions and Types

What is a good language to write proofs? Following a tradition that dates back to the *Automath* project [34], one can write the proofs and programs in the same formalism: *typed λ-calculus*. This formalism, invented by Church [24], is one of the many formalisms that can be used to describe computable algorithms and directly inspired the design of all programming languages in the *ML* family. The *Coq* system uses a very expressive variation on typed λ-calculus, the *Calculus of Inductive Constructions* [28, 70].[2]

Chapter 3 of this book covers the relation between proofs and programs, generally called the *Curry–Howard isomorphism.* The relation between a program and its type is the same as the relation between a proof and the statement it proves. Thus verifying a proof is done by a type verification algorithm. Throughout this book, we shall see that the practice of *Coq* is made easier thanks to the double knowledge mixing intuitions from functional programming and from reasoning practice.

An important characteristic of the Calculus of Constructions is that every type is also a term and also has a type. The type of a proposition is called `Prop`. For instance, the proposition $3 \leq 7$ is at the same time the type of all proofs that 3 is smaller than 7 and a term of type `Prop`.

In the same spirit, a *predicate* makes it possible to build a parametric proposition. For instance, the predicate "to be a prime number" enables us to form propositions: "7 is a prime number," "1024 is a prime number," and so on. This predicate can then be considered as a function, whose type is `nat`$\rightarrow$`Prop` (see Chap. 4). Other examples of predicates are the predicate "to be a sorted list" with type `(list Z)`$\rightarrow$`Prop` and the binary relation $\leq$, with type `Z`$\rightarrow$`Z`$\rightarrow$`Prop`.

More complex predicates can be described in *Coq* because arguments may themselves be predicates. For instance, the property of being a transitive relation on `Z` is a predicate of type `(Z`$\rightarrow$`Z`$\rightarrow$`Prop)`$\rightarrow$`Prop`. It is even possible to consider a *polymorphic* notion of transitivity with the following type:

$(A\rightarrow A\rightarrow$`Prop`$)\rightarrow$`Prop` *for every data type A.*

[2] We sometimes say *Calculus of Constructions* for short.

1.4 Specifications and Certified Programs

The few examples we have already seen show that propositions can refer to data and programs.

The *Coq* type system also makes it possible to consider the converse: the type of a program can contain constraints expressed as propositions that must be satisfied by the data. For instance, if n has type `nat`, the type "a prime number that divides n" contains a *computation-related* part "a value of type `nat`" and a *logical* part "this value is prime and divides n."

This kind of type, called a *dependent* type because it depends on n, contributes to a large extent to the expressive power of the *Coq* language. Other examples of dependent types are data structures containing size constraints: vectors of fixed length, trees of fixed height, and so on.

In the same spirit, the type of functions that map any $n > 1$ to a prime divisor of n can be described in *Coq* (see Chap. 9). Functions of this type all compute a prime divisor of the input as soon as the input satisfies the constraint. These functions can be built with the interactive help of the *Coq* system. It is called a certified program and contains both computing information and a proof: that is, how to compute such a prime divisor and the reason why the resulting number actually is a prime number dividing n.

An *extraction* algorithm makes it possible to obtain an *OCAML* [23] program that can be compiled and executed from a certified program. Such a program, obtained mechanically from the proof that a specification can be fulfilled, provides an optimal level of safety. The extraction algorithm works by removing all logical arguments to keep only the description of the computation to perform. This makes the distinction between computational and logical information important. This extraction algorithm is presented in Chaps. 10 and 11.

1.5 A Sorting Example

In this section, we informally present an example to illustrate the use of *Coq* in the development of a certified program. The reader can download the complete *Coq* source from the site of this book [10], but it is also given in the appendix at the end of this book. We consider the type "`list Z`" of the lists of elements of type `Z`. We (temporarily) use the following notation: the empty list is written `nil`, the list containing 1, 5, and -36 is written `1::5::-36::nil`, and the result of adding n in front of the list l is written $n :: l$.

How can we specify a sorting program? Such a program is a function that maps any list l of the type "`list Z`" to a list l' where all elements are placed in increasing order and where all the elements of l are present, also respecting the number of times they occur. Such properties can be formalized with two predicates whose definitions are described below.

1.5.1 Inductive Definitions

To define the predicate "to be a sorted list," we can use the *Prolog* language as inspiration. In *Prolog*, we can define a predicate with the help of clauses that enumerate sufficient conditions for this predicate to be satisfied. In our case, we consider three clauses:

1. the empty list is sorted,
2. every list with only one element is sorted,
3. if a list of the form $n :: l$ is sorted and if $p \leq n$ then the list $p :: n :: l$ is sorted.

In other words, we consider the smallest subset X of "`list Z`" that contains the empty list, all lists with only one element, and such that if the list $n :: l$ is in X and $p \leq n$ then $p :: n :: l \in X$. This kind of definition is given as an *inductive definition* for a predicate `sorted` using three constructing rules (corresponding to the clauses in a *Prolog* program).

Inductive sorted:`list Z`$\rightarrow$`Prop`:=
sorted0: sorted(nil)
sorted1: $\forall z :$ `Z`, sorted($z :: nil$)
sorted2: $\forall z_1, z_2$: `Z`, $\forall l$: `list Z`, $z_1 \leq z_2 \Rightarrow$ sorted($z_2 :: l$) $\Rightarrow$ sorted($z_1 :: z_2 :: l$)

This kind of definition is studied in Chaps. 8 and 14.

Proving, for instance, that the list `3::6::9::nil` is sorted is easy thanks to the construction rules. Reasoning about arbitrary sorted lists is also possible thanks to associated lemmas that are automatically generated by the *Coq* system. For instance, techniques known as *inversion techniques* (see Sect. 8.5.2) make it possible to prove the following lemma:

sorted_inv: $\forall z :$ `Z`, $\forall l :$ `list Z`, sorted($n :: l$) $\Rightarrow$ sorted(l)

1.5.2 The Relation "to have the same elements"

It remains to define a binary relation expressing that a list l is a permutation of another list l'. A simple way is to define a function `nb_occ` of type "`Z`$\rightarrow$`list Z`$\rightarrow$`nat`" which maps any number z and list l to the number of times that z occurs in l. This function can simply be defined as a recursive function. In a second step we can define the following binary relation on lists of elements in `Z`:

$$l \equiv l' \Leftrightarrow \forall z : \texttt{Z}, \texttt{nb\_occ}\ z\ l = \texttt{nb\_occ}\ z\ l'$$

This definition does not provide a way to determine whether two lists are permutations of each other. Actually, trying to follow it naïvely would require comparing the number of occurrences of z in l and l' for all members of $\mathbb{Z}$ and this set is infinite! Nevertheless, it is easy to prove that the relation $\equiv$ is an equivalence relation and that it satisfies the following properties:

equiv_cons: $\forall z : \texttt{Z},\ \forall l, l' : \texttt{list Z},\ l \equiv l' \Rightarrow z :: l \equiv z :: l'$
equiv_perm: $\forall n, p : \texttt{Z},\ \forall l, l' : \texttt{list Z},\ l \equiv l' \Rightarrow n :: p :: l \equiv p :: n :: l'$

These lemmas will be used in the certification of a sorting program.

1.5.3 A Specification for a Sorting Program

All the elements are now available to specify a sorting function on lists of integers. We have already seen that the *Coq* type system integrates complex specifications, with which we can constrain the input and output data of programs. The specification of a sorting algorithm is the type `Z_sort` of the functions that map any list l : `list Z` to a list l' satisfying the proposition $\mathrm{sorted}(l') \wedge l \equiv l'$.

Building a certified sorting program is the same as building a term of type `Z_sort`. In the next sections, we show how to build such a term.

1.5.4 An Auxiliary Function

For the sake of simplicity, we consider insertion sort. This algorithm relies on an auxiliary function to insert an element in an already sorted list. This function, named `aux`, has type "`Z→list Z→list Z`." We define `aux` n l in the following manner, in a recursion where l varies:

- **if** l is empty, **then** $n :: nil$,
- **if** l has the form $p :: l'$ **then**
  - **if** $n \leq p$ **then** $n :: p :: l'$,
  - **if** $p < n$, **then** $p ::$ (`aux` n l).

This definition uses a comparison between n and p. It is necessary to understand that the possibility to compare two numbers is a property of `Z`: the order $\leq$ is decidable. In other words, it is possible to program a function with two arguments n and p that returns a certain value when $n \leq p$ and a different value when $n > p$. In the *Coq* system, this property is represented by a certified program given in the standard library and called `Z_le_gt_dec` (see Sect. 9.1.3). Not every order is decidable. For instance, we can consider the type `nat→nat` representing the functions from $\mathbb{N}$ to $\mathbb{N}$ and the following relation:

$$f < g \Leftrightarrow \exists i \in \mathbb{N},\ f(i) < g(i) \wedge (\forall j \in \mathbb{N}.\ j < i \Rightarrow f(j) = g(j))$$

This order relation is undecidable and it is impossible to design a comparison program similar to `Z_le_gt_dec` for this order. A consequence of this is that we cannot design a program to sort lists of functions.[3] The purpose of function `aux` is described in the following two lemmas, which are easily proved by induction on l:

[3] This kind of problem is not inherent to *Coq*. When a programming language provides comparison primitives for a type A it is only because comparison is decidable in this type. The *Coq* system only underlines this situation.

aux_equiv: $\forall l:$ `list Z`, $\forall n:$ `Z`, `aux n l` $\equiv n :: l$,
aux_sorted: $\forall l:$ `list Z`, $\forall n:$ `Z`, `sorted` $l \Rightarrow$ `sorted aux` $n\ l$

1.5.5 The Main Sorting Function

It remains to build a certified sorting program. The goal is to map any list l to a list l' that satisfies `sorted` $l' \wedge l \equiv l'$.

This program is defined using induction on the list l:

- If l is empty, then $l' =$ `[]` is the right value.
- Otherwise l has the form $l = n :: l_1$.
  - The induction hypothesis on l_1 expresses that we can take a list l'_1 satisfying "`sorted` $l'_1 \wedge l_1 \equiv l'_1$".
    Now let l' be the list `aux` $n\ l'_1$
  - thanks to the lemma `aux_sorted` we know `sorted` l',
  - thanks to the lemma `aux_equiv` and `equiv_cons` we know

$$l = n :: l_1 \equiv n :: l'_1 \equiv \texttt{aux}\ n\ l'_1 = l'.$$

This construction of l' from l, with its logical justifications, is developed in a dialogue with the *Coq* sytem. The outcome is a term of type `Z_sort`, in other words, a certified sorting program. Using the extraction algorithm on this program, we obtain a functional program to sort lists of integers. Here is the output of the `Extraction` command:[4]

```
let rec aux z0 = function
  | Nil -> Cons (z0, Nil)
  | Cons (a, l') ->
      (match z_le_gt_dec z0 a with
         | Left -> Cons (z0, (Cons (a, l')))
         | Right -> Cons (a, (aux z0 l')))

let rec sort = function
  | Nil -> Nil
  | Cons (a, tl) -> aux a (sort tl)
```

This capability to construct mechanically a program from the proof that a specification can be satisfied is extremely important. Proofs of programs that could be done on a blackboard or on paper would be incomplete (because they are too long to write) and even if they were correct, the manual transcription into a program would still be an occasion to insert errors.

[4] This program uses a type with two constructors, `Left` and `Right`, that is isomorphic to the type of boolean values.

1.6 Learning *Coq*

The *Coq* system is a computer tool. To communicate with this tool, it is mandatory to obey the rules of a precise language containing a number of commands and syntactic conventions. The language that is used to describe terms, types, proofs, and programs is called *Gallina* and the command language is called *Vernacular*. The precise definition of these languages is given in the *Coq* reference manual [81].

Since *Coq* is an interactive tool, working with it is a dialogue that we have tried to transcribe in this book. The majority of the examples we give in this book are well-formed examples of using *Coq*. For pedagogical reasons, some examples also exhibit erroneous or clumsy uses and guidelines to avoid problems are also described.

The *Coq* development team also maintains a site that gathers all the contributions of users[5] with many formal developments concerning a large variety of application domains. We advise the reader to consult this repository regularly. We also advise suscribing to the `coq-club` mailing list,[6] where general questions about the evolution of the system appear, its logical formalism are discussed, and new user contributions are announced.

As well as for training on the *Coq* tool, this book is also a practical introduction to the theoretical framework of type theory and, more particularly, the Calculus of Inductive Constructions that combines several of the recent advances in logic from the point of view of λ-calculus and typing. This research field has its roots in the work of Russell and Whitehead, Peano, Church, Curry, Prawitz, and Aczel; the curious reader is invited to consult the collection of papers edited by J. van Heijenoort "From Frege to Gödel" [84].

1.7 Contents of This Book

The Calculus of Constructions

Chapters 2 to 4 describe the Calculus of Constructions. Chapter 2 presents the simply typed λ-calculus and its relation with functional programming. Important notions of terms, types, sorts, and reductions are presented in this chapter, together with the syntax used in *Coq*.

Chapter 3 introduces the logical aspects of *Coq*, mainly with the Curry–Howard isomorphism; this introduction uses the restricted framework that combines simply typed λ-calculus and minimal propositional logic. This makes it possible to introduce the notion of *tactics*, the tools that support interactive proof development.

The full expressive power of the Calculus of Constructions, encompassing polymorphism, dependent types, higher-order types, and so on, is studied

[5] `http://coq.inria.fr/contribs-eng.html`
[6] `coq-club@pauillac.inria.fr`

2

Types and Expressions

One of the main uses of *Coq* is to certify and, more generally, to reason about programs. We must show how the *Gallina* language represents these programs. The formalism used in this chapter is a simply typed λ-calculus [24], akin to a purely functional programming language without polymorphism. This simple formalism is introduced in a way that makes forthcoming extensions natural. With these extensions we can not only reason logically, but also build complex program specifications. To this end, classical notions like *environments*, *contexts*, *expressions* and *types* will be introduced, but we shall also see more complex notions like *sorts* and *universes.*

This chapter is also the occasion for first contact with the *Coq* system, so that we can learn the syntax of a few commands, with which we can check types and evaluate expressions.

The first examples of expressions that we present use types known by all programmers: natural numbers, integers, boolean values. For now we only need to know that the description of these types and their properties relies on techniques introduced in Chap. 6. To provide the reader with simple and familiar examples, we manipulate these types in this chapter as if they were predefined. Finally, introducing the notion of *sort* will make it possible to consider arbitrary types, a first step towards polymorphism.

2.1 First Steps

Our first contact with *Coq* is with the coq toplevel, using the command `coqtop`. The user interacts with the system with the help of a language called the *Coq* vernacular. Note that every command must terminate with a period.

The short set of commands that follows presents the command `Require`, whose arguments are a flag (here `Import`) and the name of a module or a library to load. Libraries contain definitions, theorems, and notation. In these examples, libraries deal with natural number arithmetic, integer arithmetic, and boolean values. Loading these libraries affects a component of *Coq* called the *global environment* (in short *environment*), a kind of table that keeps track of declarations and definitions of constants.

```
machine prompt %  coqtop
 Welcome to Coq 8.0 (Oct 2003)
 Require Import Arith.
 Require Import ZArith.
 Require Import Bool.
```

2.1.1 Terms, Expressions, Types

The notion of *term* covers a very general syntactic category in the *Gallina* specification language and corresponds to the intuitive notion of a well-formed expression. We come back to the rules that govern the formation of terms later. In this chapter, we mainly consider two kinds of terms, *expressions*, which correspond approximately to programs in a functional programming language, and *types*, which make it possible to determine when terms are well-formed and when they respect their specifications. Actually, the word *specification* will sometimes be used to describe the type of a program.

2.1.2 Interpretation Scopes

Mathematics and computer science rely a lot on conventional notations, often with infix operators. To simplify the input of expressions, the *Coq* system provides a notion of *interpretation scopes* (in short *scopes*), which define how notations are interpreted. Interpretation scopes usually indicate the function that is usually attached to a given notation. For instance, the infix notation with a star * can be used both in arithmetic to denote multiplication and in type languages to denote the cartesian product.

The current terminology is that scopes may be *opened* and several scopes may be opened at a time. Each scope gives the interpretation for a set of notations. When a given notation has several interpretations, the most recently opened scope takes precedence, so that the collection of opened scopes may be viewed as a stack. The command to open the scope *s* is "`Open Scope` *s*." The way to know which interpretations are valid for a notation is to use the `Locate` command. Here is an example:

```
Open Scope Z_scope.

Locate "_ * _".
...
 "x * y" := prod x y : type_scope
 "x * y" := Ring_normalize.Pmult x y : ring_scope
 "x * y" := Pmult x y : positive_scope
 "x * y" := mult x y : nat_scope
 "x * y" := Zmult x y : Z_scope (default interpretation)
```

This dialogue shows that, by default, the notation "`x * y`" will be understood as the application of the function `Zmult` (the multiplication of integers), as provided by the scope `Z_scope`.

More information about a scope *s* is obtained with the command

```
Print Scope s.
```

For instance, we can discover all the notations defined by this scope:

```
Print Scope Z_scope.
Scope Z_ scope
Delimiting key is Z
Bound to class Z
"- x" := Zopp x
"x * y" := Zmult x y
"x + y" := Zplus x y
"x - y" := Zminus x y
"x / y" := Zdiv x y
"x < y" := Zlt x y
"x < y < z" := and (Zlt x y)(Zlt y z)
"x < y <= z" := and (Zlt x y)(Zle y z)
"x <= y" := Zle x y
"x <= y < z" := and (Zle x y)(Zlt y z)
"x <= y <= z" := and (Zle x y)(Zle y z)
"x > y" := Zgt x y
"x >= y" := Zge x y
"x ?= y" := Zcompare x y
"x ^ y" := Zpower x y
"x 'mod' y" := Zmod x y
```

The *delimiting key* associated with a scope is useful to limit a scope to an expression inside a larger expression. The convention is to write the expression first, surrounded by parentheses if it is non-atomic, followed by the character %, then followed by the key. With delimiting keys, we can use several notation conventions in a single command, for instance when this expression contains both integers and real numbers.

2.1.3 Type Checking

We can use the command "`Check` *t*" to decide whether a term *t* is well-formed and what is its type. This type-checking is done with respect to an environment, determined by the declarations and definitions that were executed earlier. When the term is not well-formed, an appropriate error message is displayed.

Natural Numbers

The type of natural numbers is called `nat`, zero is actually described by the identifier `O` (the capital O letter, not the digit), and there is a function `S` that takes as argument a natural number and returns its successor. Thanks to notational conventions provided in the scope `nat_scope`, natural numbers can also be written in decimal form when this scope has precedence over the others. In practice, the natural number n is written n`%nat` outside the scope `nat_scope` and n inside this scope.

```
Check 33%nat.
33%nat : nat

Check 0%nat.
0%nat : nat

Check O.
0%nat : nat

Open Scope nat_scope.

Check 33.
33 : nat

Check 0.
0 : nat
```

Integers

The type `Z` is associated with integers: the $\mathbb{Z}$ set that is commonly used in mathematics and closely related to the type `int` in many programming languages. The library `ZArith` provides the scope `Z_scope`, so that we can write integer numbers by giving their decimal representation. As with natural numbers, integers are written with the suffix `%Z` when the most recently opened scope would give another interpretation and without the suffix when the most recently opened scope is `Z_scope`. At the beginning of the following session, the most recently opened scope is `nat_scope`.

```
Check 33%Z.
33%Z : Z

Check (-12)%Z.
(-12)%Z : Z

Open Scope Z_scope.
```

```
Check (-12).
```
-12 : Z

```
Check (33%nat).
```
33%nat : nat

In *Coq*'s type system, there is no type inclusion: a natural number is not an integer and converting one number into the other is only done with the help of explicit conversion functions.[1]

Boolean Values

The type `bool` contains two constants associated with truth values:

```
Check true.
```
true : bool

```
Check false.
```
false : bool

2.2 The Rules of the Game

In this section, we present the rules to construct well-formed terms in a subset of *Gallina* that corresponds to the simply typed λ-calculus. These rules together give the syntax of terms (types and expressions) and the constraints that make it possible to determine whether an expression respects the type discipline. We also introduce the notions of variables, constants, declarations, and definitions.

2.2.1 Simple Types

A simple framework to start our study of *Coq* is provided by the simply typed λ-calculus without polymorphism, a model of programming languages with reduced expressive power. Types have two forms:

1. *Atomic* types, made of single identifiers, like `nat`, `Z`, and `bool`.
2. Types of the form[2] $(A{\rightarrow}B)$, where A and B are themselves types. For now, we call these types *arrow* types. Arrow types represent types of

[1] Nevertheless, *Coq* provides the user with a system of *implicit coercions*. Refer to the reference manual.

[2] Note the first use of a convention that we use often in this book: terms or commands, respecting the syntax of the *Coq* input language, but where variables in italics represent arbitrary expressions; these variables are often called "meta-variables" in the computer science literature, to distinguish them from the variables of the language being described. Here $A{\rightarrow}B$ denotes the infinity of *Coq* types where A and B could be replaced by other types.

context contain declarations for disjoint sets of identifiers. The *Coq* system enforces these constraints by giving different internal names to global and local variables and by producing an error message when global or local declarations are repeated for the same identifier.

Notation

The notations introduced here do not deal directly with *Gallina*, but they are needed to describe some of the well-formedness rules and *Coq*'s behavior. Our notation is simplified; for a precise and complete formalization, readers should refer to the description of the Calculus of Inductive Constructions in the *Coq* reference manual.

Environments: In our mathematical formulas, we use the symbol E (with possible alterations and subscripts) to designate arbitrary environments.
Contexts: In a similar way as for environments, the symbol Γ is used to designate arbitrary contexts.
Empty context: We use the notation "[]" to denote the context where no local variables are declared. In particular, this is the current context when the current point is outside any section.
Declarations: The declaration which specifies that the identifier v has type A is written $(v : A)$.
Declaration sequences: A context usually appears as a sequence of declarations, presented as below:

$$[v_1 : A_1; v_2 : A_2; \ldots; v_n : A_n]$$

Adding a declaration $(v : A)$ to a context Γ is denoted $\Gamma :: (v : A)$.
Existence of a declaration: To express that a variable v is declared with type A in a context Γ, we use the notation $(v : A) \in \Gamma$; variants are also used: $v \in \Gamma$ (without detailing what is its type), $v \in E \cup \Gamma$ (the declaration is either global or local), etc.
Typing judgment: The notation $E, \Gamma \vdash t : A$ can be read "in the environment E and the context Γ, the term t has type A."
E$_0$: Especially for this chapter, we denote E_0 as the environment obtained after loading the libraries `Arith`, `ZArith`, and `Bool`.

Definition 1 (Inhabited types). *A type, A, is* inhabited *in an environment E and a context Γ if there exists a term t such that the* $E, \Gamma \vdash t : A$ *holds.*

2.2.3 Expressions and Their Types

In the same manner that types can be built from atomic types using the arrow construct, expressions can be built from variables and constants (denoted by identifiers) using a few constructs.

Identifiers

The simplest form of an expression is an identifier x. Such an expression is well-formed only if x is declared in the current environment or context. If A is the type of x in its declaration then x has type A.

This typing rule is usually presented as an *inference rule*: premises are placed above a horizontal bar while the conclusion is placed below that bar:

$$\textbf{Var}\,\frac{(x,A)\in E\cup\Gamma}{E,\Gamma\vdash x:A}.$$

This rule can be read as: "if the identifier x appears with type A in the environment E or the context Γ, then x has type A in this environment and context." It is applied in the examples of Sect. 2.1.3 for the identifiers `0:nat`, `true:bool`, and `false:bool`.

Other examples are given below using the addition functions for natural numbers and integers, and using negation and disjunction on boolean values.

```
Check plus.
```
plus : nat→nat→nat

```
Check Zplus.
```
Zplus : Z→Z→Z

```
Check negb.
```
negb : bool→bool

```
Check orb.
```
orb : bool→bool→bool

The following dialogue shows what happens when using an identifier that was not previously declared or defined:

```
Check zero.
```
...
Error: The reference "zero" was not found in the current environment

Function Application

The main control structure of our language is the application of functions to arguments.

Let us consider an environment E and a context Γ and two expressions e_1 and e_2 with respective types $A\rightarrow B$ and A in $E\cup\Gamma$; then the application of e_1 to e_2 is the term written "$e_1\ \ e_2$" and this term has type B in the environment and context being considered.

In the expression "$e_1\ \ e_2$", e_1 is said to be in the *function position*, while e_2 is the *argument*. The presentation as an inference rule is as follows:

$$\textbf{App}\ \frac{E,\Gamma \vdash e_1 : A \rightarrow B \quad E,\Gamma \vdash e_2 : A}{E,\Gamma \vdash e_1\ e_2 : B}$$

For example, in the environment E_0 (see Sect. 2.2.2), we can use the identifiers `true` and `negb` to construct a new well-formed expression and determine its type; this process can be repeated to construct more and more complex expressions:

```
Check negb.
negb : bool→bool

Check (negb true).
negb true : bool

Check (negb (negb true)).
negb (negb true) : bool
```

2.2.3.1 Syntactic Conventions

The definition of application and the typing rule only consider functions with one argument. In fact, a function with several arguments is simply represented as a function with one argument that returns another function. We have seen that a parenthesis-free notation was provided for the type of this kind of function. A similar convention appears when constructing applications of a function to several arguments. We shall write "$f\ t_1\ \ldots\ t_n$" instead of "$(f\ t_1)\ \ldots\ t_n$" thus reducing drastically the number of parentheses used. The *Coq* system automatically respects these conventions and suppresses extraneous parentheses:

```
Check (((ifb (negb false)) true) false).
ifb (negb false) true false : bool
```

However, we should be careful to keep parentheses when they are needed to ensure that the term constructed will be well-formed. The following example shows that removing too many pairs of parentheses leads to a badly formed term:

```
Check (negb negb true).
Error: The term "negb" has type "bool→bool"
 while it is expected to have type "bool"
```

The syntactic conventions for writing arrow types and applications go hand in hand to give the users the impression they are manipulating functions with several arguments. This can be summarized with a derived typing rule:

$$\textbf{App*}\ \frac{E,\Gamma \vdash e : A_1 \rightarrow A_2 \rightarrow \ldots \rightarrow A_n \rightarrow B \quad E,\Gamma \vdash e_i : A_i\ \ (i=1\ldots n)}{E,\Gamma \vdash e\ e_1\ e_2\ \ldots\ e_n : B}$$

With the help of syntactic notations and interpretation scopes, we can avoid the uniform notation of function application and rely on conventions that are closer to mathematical and programming practice. In the following, we enumerate some of the notation.

Natural numbers

All natural numbers are obtained by the repetitive application of a successor function, called `S`, to the number zero, represented by the capital letter `O`. Thus, the number n would normally be written as follows:

$$\underbrace{\texttt{S(S(S(...(S(}}_{n}\texttt{O))...))}.$$

In the scope `nat_scope`, this number is simply represented by its decimal value

```
Open Scope nat_scope.

Check (S (S (S O))).
3 : nat
```

This scope also supports the infix operations `+`, `-`, and `*` to represent the binary functions `plus`, `minus`, and `mult`.

```
Check (mult (mult 5 (minus 5 4)) 7).
5*(5-4)*7 : nat

Check (5*(5-4)*7).
5*(5-4)*7 : nat
```

The decimal and infix notation for natural numbers and operations is only a notation: each number really is a term obtained by applying the function `S` to another number and the operations are applications of binary functions, as shown in the following examples:

```
Unset Printing Notations.
Check 4.
S (S (S (S O))) : nat

Check (5*(5-4)*7).
mult (mult (S (S (S (S (S O)))))
           (minus (S (S (S (S (S O))))) (S (S (S (S O))))))
   (S (S (S (S (S (S (S O)))))))
   : nat

Set Printing Notations.
Check (minus (S (S (S (S (S O))))) (S (S (S (S O))))).
 5 - 4 : nat
```

Integers

The scope `Z_scope` is similar to `nat_scope`, with addition, multiplication, and subtraction operations actually representing the functions `Zplus`, `Zmult`, `Zminus`. There is also a prefix - sign, representing the unary `Zopp` function.

```
Open Scope Z_scope.
Check (Zopp (Zmult 3 (Zminus (-5)(-8)))).
-(3*(-5--8)) : Z

Check ((-4)*(7-7)).
 -4*(7-7) : Z
```

Examples

The dialogue shown in this section illustrates the rules **Var** and **App**. Note that the *Coq* system chooses the most concise notation when printing terms; also, functions with several arguments, like `plus` and `Zplus`, can be applied to only one argument, thus yielding new functions. In the second example, the function `Zplus` expects an integer and the scope `Z_scope` is automatically opened to read the argument given to this function; the same occurs in a later example with the function `Zabs_nat`, which expects an integer. In that example, the decimal 5 is read twice to yield two different values: a natural number (the first argument to natural number addition) and an integer (the first argument to integer subtraction).

```
Open Scope nat_scope.

Check (plus 3).
plus 3 : nat→nat

Check (Zmult (-5)).
Zmult (-5) : Z→Z

Check Zabs_nat.
Zabs_nat : Z→nat

Check (5 + Zabs_nat (5-19)).
 5 + Zabs_nat (5-19) : nat
```

In the following example, the term "`mult 3`" has type `nat→nat` and cannot take as argument the value `(-45)%Z` that has type `Z`. This violation of typing rules makes *Coq* emit an error message:

```
Check (mult 3 (-45)%Z).
Error: The term "-45%Z" has type "Z" while it is expected to have type "nat"
```

Anonymous Variables

Some abstractions are used to represent constant functions. This happens when an abstraction of the form `fun` v:T $\Rightarrow t$ is such that the variable v does not occur in t. In such cases, it is possible to use an anonymous variable instead of a fully-fledged identifier; the anonymous variable is always written using the special symbol "_."

```
Check (fun n _:nat ⇒ n).
fun n _:nat ⇒ n  : nat→nat→nat
```

The *Coq* system automatically replaces formal parameters by anonymous variables when it detects that these parameters are not used in the abstraction's body, as can be seen in the following example:

```
Check (fun n p:nat ⇒ p).
fun _ p:nat ⇒ p : nat→nat→nat
```

2.2.3.3 Local Bindings

The local binding (called `let-in` in *Coq*'s reference manual) is a construct inherited from the languages of the *Lisp* and *ML* families. It avoids repeated code and computation, by using local variables to store intermediate results.

A local binding is written `let` v`:=`t_1 `in` t_2, where v is an identifier and t_1 and t_2 are expressions. This construct is well-typed in a given environment E and context Γ when t_1 is well-typed of type A in the environment E and the context Γ and when t_2 is well-typed in the same environment and the augmented context $\Gamma :: (v : A)$. This is expressed by the following typing rule:

$$\textbf{Let-in}\ \frac{E,\Gamma \vdash t_1 : A \quad E,\Gamma :: (v := t_1 : A) \vdash t_2 : B}{E,\Gamma \vdash \texttt{let}\ v\texttt{:=}t_1\ \texttt{in}\ t_2 : B}$$

To illustrate the use of this construct, we give a representation of the function $\lambda n\,p\,.\,(n-p)^2((n-p)^2+n)$ with shared subterms:

```
fun n p : nat ⇒
   (let diff := n-p in
    let square := diff*diff in
        square * (square+n))%nat
```

Exercise 2.4 *How many instances of the typing rules are needed to type this expression?*

2.2.4 Free and Bound Variables; α-conversion

Variable binding is a very common notion in mathematics and functional programming; while we will not go into detail we give a few reminders.

The constructs "`fun` v`:`A $\Rightarrow e$" and "`let` v`:=`e_1 `in` e_2" introduce variable bindings; the scope of the bound variable v is e in the first case and e_2 in the second case; the occurrence of variable v in a term is *free* if it is not in the scope of a binding for v and bound otherwise.

For example, let us consider the term t_1 below:

```
Definition t₁ :=
 fun n:nat ⇒ let s := plus n (S n) in mult n (mult s s).
```

All occurrences of `S`, `plus`, and `mult` are free, the occurrences of `n` are bound by the abstraction "`fun n:nat ⇒...`," and the occurrences of `s` are bound by the local binding "`let s:=(plus n (S n)) in ....`"

As in logic and mathematics, the name of a bound variable can be changed in an abstraction, provided all occurrences of the bound variables are replaced in the scope, thus changing from " `fun` v`:`A $\Rightarrow t$ " to " `fun` v'`:`A $\Rightarrow t'$ " where t' is obtained by replacing all free occurrences of v in t' by v', *provided v' does not occur free in t and t contains no term of the form "* `fun` v'`:`B $\Rightarrow t''$ *" such that v appears in t'' and no term of the form "* `let` v'`:=`t''_1 `in` t''_2 *" such that v appears in t''_2*. This formulation is quite complex, but the user rarely needs to be concerned with it, because the *Coq* system takes care of this aspect. Renaming bound variables is called *α-conversion*; such a transformation applies similarly to local bindings. For instance, the following term is obtained from the previous one with α-conversion:

```
fun i : nat ⇒
   let sum := plus i (S i) in mult i (mult sum sum).
```

While we did an α-conversion above renaming `n` to `i` and `s` to `sum`, it is not possible to rename `s` to `n` (without renaming the out bound variable) because `n` occurs free inside the original body of the binding of `s`, the term `(mult n (mult s s))`. Thus, the following term is clearly not α-convertible to the value of t_1:

```
fun n : nat ⇒
  let n := plus n (S n) in mult n (mult n n).
```

The α-conversion is a congruence on the set of terms, denoted $\cong_\alpha$. In other words, this relation is an equivalence relation that is compatible with the term structure, as expressed by the following formulas:

if $t_1 \cong_\alpha t'_1$ **and** $t_2 \cong_\alpha t'_2$, **then** $t_1\ t_2 \cong_\alpha t'_1\ t'_2$,
if $t \cong_\alpha t'$, **then** `fun` v`:`A $\Rightarrow t \cong_\alpha$ `fun` v`:`A $\Rightarrow t'$,
if $t_1 \cong_\alpha t'_1$ **and** $t_2 \cong_\alpha t'_2$, **then** `let` v`:=`t_1 `in` $t_2 \cong_\alpha$ `let` v`:=`t'_1 `in` t'_2.

In the rest of this book, we consider that two α-convertible terms are equal.

2.3 Declarations and Definitions

At any time during a *Coq* session, the current environment combines the contents of the initial environment, the loaded libraries, and all the global definitions and declarations made by the user. In this section, we study the ways to extend the environment with new declarations and definitions. We first present global definitions and declarations which modify the environment. We then describe the section mechanism and local definitions and declarations which modify the context. With these commands, we can also describe parametric expressions, expressions that can be reused in a variety of situations.

2.3.1 Global Declarations and Definitions

A global declaration is written "`Parameter` v:A" and variants are also provided to declare several variables at once. The effect of this declaration is simply to add $(v : A)$ to the current environment. For instance, one may wish to work on a bounded set of integers and to declare a parameter of type `Z`:

```
Parameter max_int : Z.
```
max_int is assumed

No value is associated with the identifier `max_int`. This characteristic is permanent and there will be no way to choose a value for this identifier afterwards, as opposed to what is usual in programming languages like *C*. The constant `max_int` remains an arbitrary constant for the rest of the development.[5]

The definition of a global constant is written "`Definition` c:A`:=` t." For this definition to be accepted, it is necessary that t is well-typed in the current environment and context, that its type is A, and that c does not clash with the name of another global variable. If one wants to let the system determine the type of the new constant, one can simply write "`Definition` c `:=` t." The effect of this definition is to add $c := t : A$ to the current environment.

In the following definition we define the constant `min_int` with type `Z`. This definition uses the parameter `max_int` declared above. We use the `Print` command to see the value and type of a defined identifier:

```
Open Scope Z_scope.

Definition min_int := 1-max_int.
Print min_int.
```
min_int = 1-max_int : Z

When defining functions, several syntactic forms may be used, relying either on abstraction or on an explicit separation of the function parameters. Here are examples of the various forms for the same definition:

[5] Still, we shall see later that properties of this variables may also be added later to the environment or the context using axioms or hypotheses. However, there is a risk of introducing inconsistencies when adding axioms to the environment.

```
Definition cube := fun z:Z ⇒ z*z*z.

Definition cube (z:Z) : Z := z*z*z.

Definition cube z := z*z*z.
```

After any of these three variants, the behavior of `Print` is the same:

```
Print cube.
```
*cube = fun z:Z ⇒ z*z*z : Z→Z*
Argument scope is [Z_scope]

This shows that the argument given to this function will automatically be interpreted in the scope `Z_scope`.

Of course we can also define functionals and reuse them in other definitions:

```
Definition Z_thrice (f:Z→Z)(z:Z) := f (f (f z)).

Definition plus9 := Z_thrice (Z_thrice (fun z:Z ⇒ z+1)).
```

Exercise 2.5 *Write a function that takes five integer arguments and returns their sum.*

2.3.2 Sections and Local Variables

Sections define a block mechanism, similar to the one found in many programming languages (*C*, *Java*, *Pascal*, etc.). With sections, we can declare or define local variables and control their scope.

In the *Coq* system, sections have a name and the commands to start and finish a section are respectively "`Section` ***id***" and "`End` ***id***." Sections can be nested and opening and closing commands must respect the same kind of discipline as parentheses.

Here is a small sample development where the structure is given by section. The theme of this example will revolve around polynomials of degree 1 or 2. The function `binomial` is defined in a context Γ_1 where a and b are declared of type Z.

```
Section binomial_def.
 Variables a b:Z.
 Definition binomial z:Z := a*z + b.
 Section trinomial_def.
  Variable c : Z.
  Definition trinomial z:Z := (binomial z)*z + c.
 End trinomial_def.
End binomial_def.
```

In this development there are two nested sections named `binomial_def` and `trinomial_def`. The most external section is at the global level, outside every section.

The `binomial_def` section starts with the declaration of two local variables `a` and `b` of type `Z`. The keyword `Variable` indicates a local declaration, unlike the keyword `Parameter` that was used for global declarations. The scope of the declaration is limited to the rest of the `binomial_def` section. The current context is extended to add the declarations of `a` and `b`; in other words, we have a new context $\Gamma_1 = [a : Z; b : Z]$. The same happens with the local declaration of `c`, where the current context is extended with $(c : Z)$ until the end of `trinomial_def`. The new context is $\Gamma_2 = [a : Z; b : Z; c : Z]$.

The global definitions of `binomial` and `trinomial` are done in different non-empty contexts. The value associated with `binomial` is well-typed in context Γ_1 and the value associated with `trinomial` is well-typed in context Γ_2.

When a constant's value uses local variables, this value may change as the sections are closed: local variables may disappear from the current context and the constant's value would be badly typed if there were no alteration. The modification consists in adding an abstraction for every local variable used in the value and the constant's type also needs to be changed accordingly.

To illustrate this evolution, we repeat the definitions, but use `Print` command to show the value of each constant inside and outside the various sections.

```
Reset binomial_def.

Section binomial_def.
 Variables a b:Z.
 Definition binomial (z:Z):= a*z + b.
 Print binomial.
binomial = fun z:Z ⇒ a*z + b
    : Z→Z
Argument scope is [Z_scope]
 Section trinomial_def.
  Variable c : Z.
  Definition trinomial (z:Z) := (binomial z)*z + c.
  Print trinomial.
trinomial = fun z:Z ⇒ binomial z * z + c
    : Z→Z
Argument scope is [Z_scope]
 End trinomial_def.
 Print trinomial.
 trinomial = fun c z:Z ⇒ binomial z * z + c
    : Z→Z→Z
Argument scopes are [Z_scope Z_scope]
```

There are four kinds of reductions used in *Coq*, but before presenting them, we must describe the elementary operation known as *substitution.*

2.4.1 Substitution

The operation of substitution consists in replacing every occurrence of a variable by a term. This operation must be done in a way that makes sure α-conversion is still a congruence. For this reason, substitutions are often accompanied by many α-conversions.

If t and u are two terms and v a variable, we denote t{v/u} as the term obtained by replacing all free occurrences of v by u in t, with the right amount of α-conversions so that free occurrences of variables in u are still free in all copies of u that occur in the result. We say that t{v/u} is an *instance* of t.

For instance, let us consider the terms $t = A\rightarrow A$ and $u =$ `nat→nat`. The term t{A/u} is `(nat→nat)→nat→nat`. As a second example, consider $t =$ `fun z:Z ⇒z*(x+z)`, $v =$ `x`, and $u =$ `z+1`; before replacing the free occurrence of `x` in t, we perform an α-conversion on the bound variable `z` with a new variable name, say `w`. We obtain the term `fun w:Z ⇒w*((z+1)+w)`. Had we not made this α-conversion the result term would have had three occurrences of the bound variable, while there were initially two of them.

2.4.2 Reduction Rules

In this section, we present the four kinds of reduction that cover all the reductions used in the type-checker to ensure that terms are well-typed.

δ-reduction (pronounced *delta*-reduction) is used to replace an identifier with its definition: if t is a term and v an identifier with value t' in the current context, then δ-reducing v in t will transform t into $t\{v/t'\}$.

In the following examples we use δ-conversion on constants `Zsqr` and `my_fun`. The `delta` keyword is used, followed by a list of the identifiers that can be reduced. This list is optional, when it is absent, all identifiers bound to some value are reduced. The `cbv` keyword indicates that the "call-by-value" strategy is used.

```
Definition Zsqr (z:Z) : Z := z*z.


Definition my_fun (f:Z→Z)(z:Z) : Z := f (f z).

Eval cbv delta [my_fun Zsqr] in (my_fun Zsqr).
     = (fun (f:Z→Z)(z:Z) ⇒ f (f z))(fun z:Z ⇒ z*z)
     : Z→Z
```

```
Eval cbv delta [my_fun] in (my_fun Zsqr).
  = (fun (f:Z→Z)(z:Z) ⇒ f (f z)) Zsqr
    : Z→Z
```

β-reduction (pronounced *beta*-reduction) makes it possible to transform a β-redex, that is, a term of the form "`(fun` v:T ⇒e_1`)` e_2," into the term $e_1\{v/e_2\}$. The following example shows a series of β-reductions. In the second term, we can observe two β-redexes. The first one is associated with the abstraction on `f` and the second one with the abstraction on `z0`. The term of the third line is obtained by applying a "call by value" strategy.

1. `(fun (f:Z→Z)(z:Z) ⇒ f (f z))(fun (z:Z) ⇒ z*z)`

2.
   ```
   fun z:Z ⇒
       (fun z1:Z ⇒ z1*z1)((fun z0:Z ⇒ z0*z0) z)
   ```

3. `fun z:Z ⇒ (fun z1:Z ⇒ z1*z1)(z*z)`

4. `fun z:Z ⇒ z*z*(z*z).`

Note that reducing the redex on the first line actually created a new redex, associated with the abstraction on `z1`. In *Coq*, we can get the same result by requesting simultaneous β- and δ-conversions:

```
Eval cbv beta delta [my_fun Zsqr] in (my_fun Zsqr).
    = fun z:Z ⇒ z*z*(z*z) : Z→Z
```

ζ-reduction (pronounced *zeta*-reduction) is concerned with transforming local bindings into equivalent forms that do not use the local binding construct. More precisely, it replaces any formula of the form `let` v`:=`e_1 `in` e_2 with $e_2\{v/e_1\}$.

For example, let us reuse the function `h` defined in Sect. 2.3.2.1. This function was defined with the help of auxiliary locally defined values that were replaced with local bindings when exiting the section. We show how a term using this function is evaluated with and without ζ-conversion:

```
Eval cbv  beta  delta [h] in (h 56 78).
 = let s := 56+78 in let d := 56-78 in s*s + d*d
   : Z
 Eval cbv  beta zeta delta [h] in (h 56 78).
 = (56+78)*(56+78)+(56-78)*(56-78)
   :Z
```

ι-reduction (pronounced *iota*-reduction) is related to inductive objects and is presented in greater details in another part of the book (Sect. 6.3.3

and 6.1.4). For now, we simply need to know that ι-reduction is responsible for computation in recursive programs. In particular most numerical functions, like addition, multiplication, and substraction, are computed using ι-reductions. The ι-reduction is a strong enough tool to "finish" our computations with the functions `h` and `my_fun`. Note that `compute` is a synonym for `cbv iota beta zeta delta`.

```
Eval compute in (h 56 78).
   = 18440 : Z

Eval compute in (my_fun Zsqr 3).
   = 81 : Z
```

Exercise 2.7 *Write the function that corresponds to the polynomial* $2 \times x^2 + 3 \times x + 3$ *on relative integers, using* λ*-abstraction and the functions* `Zplus` *and* `Zmult` *provided in the* `ZArith` *library of* Coq. *Compute the value of this function on integers* 2, 3, *and* 4.

2.4.3 Reduction Sequences

Reductions interact with the typing rules of the Calculus of Inductive Constructions. We need notation for these conversions.

Notation

The first notation we introduce expresses the statement "in context Γ and environment E, term t' is obtained from t through a sequence of β-reductions." The notation is as follows:

$$E, \Gamma \vdash t \rhd_\beta t'.$$

If we want to consider an arbitrary combination of β-, δ-, ζ-, or ι-conversions, this is expressed with the help of the indices to the reduction symbol. For example, combining β-, δ-, and ζ-conversion is written as follows:

$$E, \Gamma \vdash t \rhd_{\beta\delta\zeta} t'.$$

Combinations of β-, δ-, ι-, and ζ-reductions enjoy very important properties:

- Every sequence of reductions from a given term is finite. In other words, every computation on a term in the Calculus of Inductive Constructions terminates. This property is called *strong normalization.*
- If t can be transformed into t_1 and t_2 (using two different sequences of reductions), then there exists a term t_3 such that t_1 and t_2 can both be reduced to t_3. This is the *confluence* property,
- If t can be reduced in t', and t has type A, then t' has type A.

An important consequence of the first two properties is that any term t has a unique normal form with respect to each of the reductions.

2.4.4 Convertibility

An important property is *convertibility*: two terms are convertible if they can be reduced to the same term, using the combination of all four reductions. This property is decidable in the Calculus of Inductive Constructions. We will write this as follows:

$$E, \Gamma \vdash t =_{\beta\delta\zeta\iota} t'$$

For instance, the following two terms are convertible, since they can both be reduced to "`3*3*3`":

1. `let x:=3 in let y:=x*x in y*x`
2. `(fun z:Z ⇒z*z*z) 3`

That convertibility is decidable is a direct consequence of the abstract properties of reduction: to decide whether t and t' are convertible, it is enough to compute their normal forms and then to compare these normal forms modulo α-conversion.

2.5 Types, Sorts, and Universes

Up to this point, we have restricted our examples to a fixed set of atomic types, `nat`, `Z`, `bool`, and combinations of these types using the arrow construct. It is important to be able to define new type names and also to be able to write functions working on arbitrary types, a first step towards polymorphism. Instead of defining new mechanisms for this, *Coq* designers have extended the mechanisms that were already present in typed λ-calculus. It is enough to consider that expressions and types are particular cases of terms, and that all notions, like typing, declarations, definitions, and the like, should be applicable to all kinds of terms, whether they are types or expressions. Thus, a new question arises:

If a type is a term, what is its type?

2.5.1 The `Set` Sort

In the Calculus of Inductive Constructions, the type of a type is called a *sort*. A sort is always an identifier.

The sort `Set` is one of the predefined sorts of *Coq*. It is mainly used to describe data types and program specifications.

Definition 2 (Specification). *Every term whose type is* `Set` *is called a* specification.

Definition 3 (Programs). *Every term whose type is a specification is called a* program.

All the types we have considered so far are specifications and, accordingly, all expressions we have considered are programs. This terminology is slightly abusive, as we consider values of type `nat`, for instance, as programs. Being a total function of type `nat→nat` is a specification: we already ensure that such a function will compute without a problem, terminate, and return a result, whenever it is given an argument of the right type. Such a specification is still very weak, but we show ways to describe richer specifications, such as "a prime number greater than 567347" or "a sorting function" in Chap. 4.

For example, we can use the command `Check` to verify that the types we have used are specifications:

```
Check Z.
Z : Set
Check ((Z→Z)→nat→nat).
(Z→Z)→nat→nat : Set
```

Since types are terms and they have a type, there must be typing rules that govern the assignment of a type to a type expression. For atomic types, there are two ways: either one simply declares a new atomic type or one defines one, for instance an inductive type as in Chap. 6. For types obtained using the arrow construct, here is a simplified form of the typing rule:

$$\textbf{Prod-Set}\ \frac{E,\Gamma \vdash A : \texttt{Set} \quad E,\Gamma \vdash B : \texttt{Set}}{E,\Gamma \vdash A{\rightarrow}B : \texttt{Set}}$$

As an example, we can get the judgment $E_0, [] \vdash$ `(Z→Z)→nat→nat : Set` by using the declarations `nat` : `Set` and `Z` : `Set` and then applying the rule above three times.

2.5.2 Universes

The `Set` sort is a term in the Calculus of Inductive Constructions and must in turn have a type, but this type—itself a term—must have another type. The Calculus of Inductive Constructions considers an infinite hierarchy of sorts called *universes*. This family is formed with types $\texttt{Type}(i)$ for every i in $\mathbb{N}$ and it satisfies the following relations:

$$\begin{array}{rl} \texttt{Set} : \texttt{Type}(i) & \text{(for every } i\text{)} \\ \texttt{Type}(i) : \texttt{Type}(j) & \text{(if } i < j\text{)} \end{array}$$

The set of terms in the Calculus of Inductive Constructions is then organized in levels. So far, we have encountered the following categories:

Level 0: programs and basic values, like `O`, `S`, `trinomial`, `my_fun`, and so on.
Level 1: specifications and data types, like `nat`, `nat→nat`, `(Z→Z)→nat→nat`, and so on.
Level 2: the `Set` sort.