Albatross - Language Description

Helmut Brandl

email: firstname dot lastname at gmx dot net

Version 0.2

Contents

Chapter 1

Introduction

Albatross is a programming language which can be verified statically. You write programs in Albatross and prove them to be correct in the same language.

What is a correct program? A program is correct if it is consistent with its specification. Specifications in Albatross are assertions which express correctness conditions. Assertions are boolean expressions in predicate logic.

A verified Albatross programm has as proof for each assertion. The proof is generated by the compiler.

But since assertions are expressed in predicate logic and predicate logic is not decidable for arbitrary expressions the theorem prover in the Albatross compiler cannot prove all valid assertions.

Therefore the programmer has to provide the proof steps which cannot be done by the compiler automatically. The automation features have been designed with the following philosophy:

- The compiler does all low level work that can be done efficiently without entering a hugh search space.
- The programmer has to provide the more intelligent steps which require some creativity.

Because of this philosophy the Albatross compiler is something between a proof assistant and a theorem prover.

Chapter 2

Overview

Chapter 3

Tutorial

3.1 Getting Started

3.1.1 A Minimal Albatross Program

In the following we assume that you have successfully installed the Albatross compiler under the name `alba` and compiled the basic library of the Albatross system and initialized an environment variable which points to the location of the base library as described in the Readme file of the albatross distribution.

An Albatross program is an albatross package which is a directory with collection of albatross source files. Since we want to create a minimal albatross programm we create a package by creating a new directory with one source file called `minimal.al`. Any text editor can be used to create the file.

Our version of the minimal program looks like

```
-- file: minimal.al

use
   alba.base.boolean   -- uses the module 'boolean' of the
                       -- library 'alba.base'
end

all(a:BOOLEAN)
    require
        a              -- assume 'a'
    ensure
        a              -- conclude 'a'
    end
```

The directory where the source file resides is not yet initialized as an albatross directory. We initialize it by issuing the command

```
alba init
```

from the command line. The albatross compiler keeps track of dependencies between modules and the init command creates a hidden subdirectory in the

package directory where the compiler can store dependencies and some other information on compiled modules.

Issuing the command

```
alba status
```

we get the information

```
new:    minimal.al
```

i.e. the compiler tells us that the file is new and has not yet been compiled.

With the command

```
alba compile
```

we compile the module successfully and a following status request shows that nothing has to be done for the package.

If we modify the file and request the status again we get the answer

```
modified:   minimal.al
```

and in the case that more modules depend on this module we also get the list of all files which need recompilation because of the modification.

Now let us look at the content of the source file. The first block of a source file is the usage block where all used modules are declared.

Our minimal program uses just the module `boolean` of the library `alba.base`. This is necessary because we want to use a variable of type `BOOLEAN` which is declared in the module `boolean` of the base library.

It is difficult to do anything without the data type `BOOLEAN` in any Albatross module. Therefore it is always used either explicitly or implicitly. The module `boolean` of the base library is the only module which doesn't use other modules.

After the use block there is one proof which proves a rather trivial fact. It declares a variable `a` of type `BOOLEAN` for the scope of the proof. Then it assumes that `a` is valid. From the assumption it concludes that `a` is valid which succeeds trivially.

After the successful proof the proof engine generates the expression

```
all(a:BOOLEAN) a ==> a
```

and adds it to the global context. It has proved `a` under the assumption `a` for an arbitrary boolean expression which is a proof of the implication `a ==> a`.

This *discharging* of the assumptions is a generic procedure of the proof engine. Whenever the proof engine can verify the assertion

```
all(x:X, y:Y,...)
    require
        a; b; ...
    proof
        ...
    ensure
        z
    end
```

it generates the discharged expression

```
all(x:X, y:Y, ...) a ==> b ==> ... ==> z
```

and adds it to the context.

Note that implication is right associative.

3.1.2 Reasoning with Implication

Implication is the most important boolean operator for doing proofs. The assertion `a ==> b` states that given `a` we can immediately conclude `b`.

We can express the modus ponens law in Albatross as

```
all(a,b:BOOLEAN)    -- modus ponens law of logic
    require
        a
        a ==> b
    ensure
        b
    end
```

This law is hardwired within the proof engine. Therefore the above assertion compiles without problems.

The implication operator is right associative i.e. `a ==> b ==> c` is parsed as `a ==> (b ==> c)`. Therefore we can prove

```
all(a,b,c:BOOLEAN)
    require
        a ==> b ==> c
        a
    ensure
        b ==> c
    end
```

in one step by applying the modus ponens law. From `a` and `a ==> b ==> c` we can conclude `b ==> c`. If we have furthermore `b` we can conclude `c`. I.e. the following compiles as well

```
all(a,b,c:BOOLEAN)
    require
        a ==> b ==> c
        b
        a
    ensure
        c
    end
```

Note that the order of the assumptions is irrelevant for the validity of the conclusion.

The Albatross compiler does a full forward closure with respect to implication so that the following assertion is accepted by the compiler as well.

```
all(a,b,c,d,x,y,z:BOOLEAN)
    require
        a ==> b ==> c
        a ==> b ==> d
        x ==> y ==> z
        c ==> x
```

```
        d ==> y
        a
        b
    ensure
        z
    end
```

Let us see how the compiler verifies this assertion. Internally the compiler generates the following detailed proof

```
all(a,b,c,d,x,y,z:BOOLEAN)
    require
        a ==> b ==> c    --  1
        a ==> b ==> d    --  2
        x ==> y ==> z    --  3
        c ==> x          --  4
        d ==> y          --  5
        a                --  6
        b                --  7
    proof  -- generated by the proof engine
        b ==> c          --  8 from 6,1
        c                --  9 from 7,8
        b ==> d          -- 10 from 6,2
        d                -- 11 from 7,10
        x                -- 12 from 9,4
        y                -- 13 from 11,5
        y ==> z          -- 14 from 13,3
    ensure
        z                -- from 13,14
    end
```

All these steps are trivial applications of the modus ponens law. But it would be very awckward if the programmer had to enter all these steps. It's better to let the compiler generate them.

3.1.3 The Deduction Law

Like the modus ponens law, the deduction law is hardwired into the proof engine.

The law states: We can prove an implication `a ==> b` by assuming `a` and deriving `b` from it. On the other hand if we can prove `b` under the assumption of `a` we can conclude the implication `a ==> b` from it.

We can demonstrate the application of the deduction law with the example of the trivial assertion `a ==> a`.

```
all(a:BOOLEAN)
    ensure
        a ==> a
    end
```

The statement "`a` follows from `a`" is intuitively evident to us. But how do we prove it? The compiler uses the deduction law and creates the following detailed proof

```
all(a:BOOLEAN)
    proof   -- generated by the proof engine
        require
```

```
            a
        ensure
            a
        end
    ensure
        a ==> a
    end
```

Since there is no direct proof of `a ==> a` the proof engine shifts the antecedent `a` of the implication into the assumptions and tries to prove the consequent `a` of the implication from it. The latter step succeeds trivially because the assumption and the goal are identical.

Don't be misled by the triviality of the example. The deduction law scales to more complicated settings and is very useful. Let's assume that we want to prove the implication chain `a ==> b ==> c ==> ... ==> z`. In order to prove the chain the proof engine tries the deduction law

```
all(a,b,c,...,z:BOOLEAN)
    proof
        require
            a; b; c; ...
        proof
            ...
        ensure
            z
        end
    ensure
        a ==> b ==> c ==> ... ==> z
    end
```

and it is very convenient that the engine does it automatically.

3.1.4 Conjunction

The interface file of the module `boolean` gives us the following statements about conjunction

```
-- file: boolean.ali
(and) (a,b:BOOLEAN): BOOLEAN

all(a,b:BOOLEAN)
    ensure
        a ==> b ==> a and b     -- introduction rule of 'and'
        a and b ==> a           -- elimination rule of 'and'
        a and b ==> b           --          "
    end
```

For the the conjunction operator only a signature is given. From it we only know that conjunction is a binary boolean operator. No definition is available.

Instead of a definition the interface file gives us one introduction rule and two elimination rules.

The introduction rule states: If we have a proof of `a` and a proof of `b` we can conclude `a and b` from it.

The elimination rules state: If we have a proof of `a and b` we can conclude either operand from it.

The introduction and elimination rules are schematic rules. For the variables `a` and `b` we can substitute any boolean expression to obtain a valid assertion.

Using the introduction and elimination rules the proof engine can prove the commutativity of `and` automatically.

```
all(a,b:BOOLEAN)
    ensure
        a and b ==> b and a
    end
```

In order to understand what happens under the hood let us look at the detailed proof which is generated by the proof engine.

```
all(a,b:BOOLEAN)
    proof   -- generated by the proof engine
        require
            a and b
        proof
            a and b ==> a
            a
            a and b ==> b
            b
            b ==> a ==> b and a
            a ==> b and a
        ensure
            b and a
        end
    ensure
        a and b ==> b and a
    end
```

The generated proof is just a straighforward application of the deduction law, the elimination and introduction rules.

Note that the proof engine applies schematic rules in forward direction only if the conclusion is strictly simpler than the premise. Since this is the case for the elimination rules of `and` they are applied automatically.

In the same manner the proof engine can prove the associativity of `and` automatically.

```
all(a,b,c:BOOLEAN)
    ensure
        a and b and c   ==> a and (b and c)
        a and (b and c) ==> a and b and c
        -- Note: 'and' is left associative
    end
```

3.1.5 Disjunction

The module `boolean` of the basic library offers the following declarations for disjunction:

```
-- file: boolean.ali
```

```
(or) (a,b:BOOLEAN): BOOLEAN

all(a,b,c:BOOLEAN)
    ensure
        a ==> a or b        -- introduction law

        b ==> a or b        --          "

        a or b              -- elimination law
        ==> (a ==> c)
        ==> (b ==> c)
        ==> c
    end
```

The operator `or` is declared without definition. It has two introduction laws which should be quite obvious and the elimination law.

The eliminiation law allows us to prove assertions by case split. Let's assume that we have the assertion `a or b` in the context and we want to prove some goal `c`. Let's further assume that we can prove the goal `c` under the assumption of `a` (i.e. `a ==> c`) and we can prove the goal `c` under the assumption of `b`. Then the goal certainly has to be valid because either `a` or `b` is valid.

What does the proof engine if it sees an disjunction of the form `a or b` in the context? It does a partial specialization of the elimination law. The specialization is partial because we have substitution values for `a` and `b` but not for `c`. The partial specialization looks like

```
a or b ==> all(c) (a ==> c) ==> (b ==> c) ==> c
```

After this it applies the modus ponens law to get

```
all(c) (a ==> c) ==> (b ==> c) ==> c
```

into the context.

Note that `a` and `b` are no longer bound variables.

3.2 Boolean Logic

The Albatross base library has a module called `boolean_logic` which provides to the user a lot of useful assertions for boolean expressions. In this chapter we try to proof some useful facts abount boolean logic without using this module.

We just use the bare bone module `boolean` which provides us with the following declarations in its interface file:

```
immutable class BOOLEAN end

false: BOOLEAN
true:  BOOLEAN
(==>) (a,b:BOOLEAN): BOOLEAN
(and) (a,b:BOOLEAN): BOOLEAN
(or)  (a,b:BOOLEAN): BOOLEAN
(not) (a:BOOLEAN):   BOOLEAN -> a ==> false
(=)   (a,b:BOOLEAN): BOOLEAN -> (a ==> b) and (b ==> a)
```

```
all(a,b,c:BOOLEAN)
    ensure
        true

        a = a

        -- negation
        (not a ==> false) ==> a       -- indirect proof

        -- conjunction
        a and b ==> a                 -- and elimination
        a and b ==> b
        a ==> b ==> a and b           -- and introduction

        -- disjunction
        a ==> a or b                  -- or introduction
        b ==> a or b

        a or b
        ==> (a ==> c)
        ==> (b ==> c)
        ==> c                         -- or elimination
    end
```

We claim that these definitions and assertions are sufficient to prove all laws of boolean logic.

All the following proofs are valid within a module with the header

```
-- file: boolean_playground.al

use alba.base.boolean end

...
```

3.2.1 Idempotence of Conjunction and Disjunction

Let us begin with the simple facts that `a and a ==> a` and `a or a ==> a` should be valid in boolean logic. The first one is proved immediately by the proof engine because it just applies the deduction rule to shift `a and a` into the context and then uses the elimination rule to conclude `a` which proves the goal. For the second one the proof engine gets stuck. It applies the deduction rule to shift `a or a` into the context and then it partially specializes the elimination rule for `or` to reach the state:

```
all(a:BOOLEAN)
    require
        a or a
    proof
        all(c) (a ==> c) ==> (a ==> c) ==> c  -- (1)
        ...  -- cannot continue
    ensure
        a
    end
```

The proof engine has done all forward reasoning it can do. No backward reasoning is possible because (1) is not a valid backward rule. Why is (1) not a valid backward rule? A valid backward rule in Albatross has to satisfy the following criteria:

1. It is an implication chain.
2. The final target of the implication chain contains all bound variables.
3. The final target is not a single variable.

The assertion (1) is an implication chain and the final target `c` contains all bound variables (there is only one in this case). However the last condition is not satisfied. But the assertion (1) is still a valid forward rule. The proof engine needs just a little hint on how to specialize the first premise of the implication chain. The expression `a ==> a` can be proved trivially (see last chapter) and does the needed specialization. The complete proof:

```
all(a:BOOLEAN)
    require
        a or a
    proof
        a ==> a
    ensure
        a
    end
```

This proof is a general pattern for exploiting disjunctions in the context. Whenever we have an expression of the form `a or b` in the context and we want to prove a goal `g` we have to prove the goal under the assumption `a` and we have to prove the goal under the assumption `b`.

3.2.2 Ex Falso Quodlibet

The latin expression *ex falso quodlibet* translates to *from a false assumption everything can be concluded*. This is a general law of boolean logic and should be provable within Albatross. If we try to prove this law the proof engine gets stuck again.

```
all(a:BOOLEAN)
    require
        false
    proof
        ...  -- cannot continue
    ensure
        a
    end
```

The proof engine cannot conclude anything from `false` because there is no implication which has `false` as a first premise. No backward reasoning starting from the target is possible because the only potential assertion

```
all(a:BOOLEAN) (not a ==> false) ==> a
```

is not a valid backward rule (the final target is a single variable). We can use it as a forward rule if we can prove its premise `not a ==> false` in the context. In order to prove this we can shift `not a` into the assumptions and prove `false`. A prove of `false` succeeds in this context, because `false` is already an assumption. The complete proof:

```
all(a:BOOLEAN)
    require
        false
    proof
        not a ==> false
    ensure
        a
    end
```

This proof demostrates a generic pattern to trigger the proof engine to prove something by contradiction. We can even shorten the proof a little bit by writing

```
all(a:BOOLEAN)
    require
        false
    proof
        not not a
    ensure
        a
    end
```

Why does this work? On trying to prove `not not a` the proof engine unfolds the definition of `not` to get the new goal `not a ==> false` which it proves as already explained. As soon as it has proved `not not a` it shifts it as proved into the context and does forward reasoning. Unfolding of function definitions is done in forward direction as well, therefore `not a ==> false` is shifted into the context. Further forward reasoning leads to `a` which proves the goal.

Let's for the moment be pedantic and list all steps done by the proof engine to prove the *ex falso quodlibet* law. In order to understand the backward reasoning steps read from `not not a` upward. In oder to understand the forward reasoning steps read from `not not a` downward.

```
all(a:BOOLEAN)
    require
        false
    proof
        require
            not a
        ensure
            false
        end              -- deduction rule

        not a ==> false  -- function expansion

        not not a        -- entered by the user

        not a ==> false  -- function expansion

        (not a ==> false) ==> a  -- indirect proof rule
```

```
        a                -- modus ponens
    ensure
        a
    end
```

3.2.3 De Morgan Laws

The De Morgan laws of logic express a certain connection between negation, conjunction and disjunction.

```
not (a or b)  = (not a and not b)   -- (1)

not (a and b) = (not a or not b)    -- (2)
```

Note the need of parentheses here because the boolean operators bind stronger than the relational operator =. These laws are boolean equivalences. From the definition of = we know immediately that for each equivalence we have to prove two implications.

In Albatross we nearly never prove boolean equivalences directly. Instead we prove the two associated implications. This has two reasons:

1. Having the two implications the proof engine can prove the equivalence easily because it justs has to expand the definition of equivalence and connect the result to the two implications (by using the introduction rule of `and`).

2. Implications are very useful for backward and forward reasoning. A boolean equivalence is nearly useless for the proof engine.

The same applies to conjunctions: Don't prove them directly; better prove the operands.

If we try to prove the first one and type

```
all(a,b:BOOLEAN)
    require
        not (a or b)
    ensure
        not a
        not b
    end
```

and start the Albatross compiler then the compiler succeeds without any complaint. Let's see the detailed steps of the proof engine to understand the success.

```
all(a,b:BOOLEAN)
    require
        not (a or b)
    proof
        a or b ==> false      -- function expansion
                              -- forward reasoning stops here

        require a
        proof   a or b        -- needed premise for 'false'
```

```
        ensure  false end      -- deduction law

        a ==> false            -- function expansion

        require b
        proof   a or b         -- needed premise for 'false'
        ensure  false end      -- deduction law

        b ==> false            -- function expansion
    ensure
        not a
        not b
    end
```

To understand the detailed steps you have to read it downwards to `a or b ==> false`. The rest you have to read from the goals upwards.

The second implication of (1) immediately succeeds as well.

```
all(a,b:BOOLEAN)
    require
        not a
        not b
    ensure
        not (a or b)
    end
```

Please try to understand the detailed steps before reading further.

For completeness we give the proofs of the two implications needed for (2).

```
all(a,b:BOOLEAN)
    require
        not (a and b)
    proof
        not not (not a  or  not b)
    ensure
        not a  or  not b
    end
```

```
all(a,b:BOOLEAN)
    require
        not a  or  not b
    proof
        not a  ==>  not (a and b)
        not b  ==>  not (a and b)
    ensure
        not (a and b)
    end
```

The first one triggers a proof by contradiction. The second one uses the elimination rule of `or` to do the job.

Now let us look at the famous *tertium non datur* or the law of the excluded middle.

```
a  or  not a
```

We can prove this by triggering a proof by contradiction.

```
all(a:BOOLEAN)
    proof
        not not (a  or  not a)
    ensure
        a  or  not a
    end
```

It is a little bit tricky to understand why this proof succeeds without more hints. By expanding the function `not` and applying the deduction rule we get `not (a or not a)` into the context with the new goal `false`. With the help of the De Morgan law (1) in forward direction we get `not a` and `not not a` into the context. Function expansion on the latter term yields `not a ==> false`. The modus ponens law concludes from `not a` and `not a ==> false` the goal.

Exercises:

1. Prove the associativity law of `(a or b) or c ==> a or (b or c)`. Hint: Use the elimination law of disjunction more than once. Then make the proof compact, i.e. throw out all intermediate steps which can be done by the proof engine automatically.

2. Prove the law `(not a ==> b) ==> a or b`. Hint: Use the law of the excluded middle.

3.3 Important Modules of the Base Library

3.3.1 The Module 'any'

The class `ANY` declared in the module `any` is the base class for nearly all classes (therefore the name), because it introduces equality.

The module `any` uses some important concepts of Albatross like deferred classes, type variables, inheritance, deferred functions and assertions.

The interface file of the module `any` contains just a handful of declarations:

```
-- file: any.ali

use boolean end

deferred class ANY end

G: ANY


(=) (a,b:G): BOOLEAN    deferred end

(/=) (a,b:G): BOOLEAN -> not (a = b)

all(a:G) deferred ensure a = a end

all(a:G) ensure a /= a ==> false end

immutable class boolean.BOOLEAN
inherit         ANY end
```

The class ANY is a deferred class i.e. no objects of type ANY can be created. A deferred class is meant to be inherited by other classes which implement the deferred features and assertions.

Only deferred classes can have deferred functions and assertions.

The second declaration is the declaration of the type variable G. The scope of a type variable is the module. A type variable has a concept. The concept of the type variable G is ANY. I.e. in any places where the type variable G occurs it can be replaced by any type which satisfies ANY i.e. which inherits from ANY.

The module declares the two functions = and /=. The function = is a deferred function which has to be present in all descendants of the class ANY, either declared in the descendant with a definition or redeclared as deferred. The function /= has a definition.

Furthermore the module declares a deferred assertion which states reflexivity of equality. Like deferred functions all deferred assertions have to be available in any descendant.

The second assertion is effective (i.e. not deferred) which states that the implementation file has a proof for this statement. This assertion is available to all descendants of ANY for free.

The last statement is an inheritance statement. It states that the class BOOLEAN of the module boolean can inherit ANY i.e. it has been successfully verified that the class BOOLEAN has a reflexive equality function and therefore is entitled to inherit from ANY the effective function /= and the effective assertion stating that irreflexivity implies falsehood.

3.3.2 The Module 'predicate'

The concept of a predicate is very important in Albatross. A predicate is a total boolean valued function over a certain type. All objects which satisfy a predicate form a set. Therefore a predicate represents a set.

Let us look into the interface file of the module predicate to explain some concepts.

```
-- file: predicate.ali

use any end

G: ANY

immutable class PREDICATE[G] end

(in)  (a:G, p:G?): BOOLEAN
(/in) (a:G, p:G?): BOOLEAN -> not p(a)

(<=) (p,q:G?): ghost BOOLEAN -> all(x) p(x) ==> q(x)
(=)  (p,q:G?): ghost BOOLEAN -> p <= q and q <= p


all(p:G?) ensure p = p end

immutable class PREDICATE[G]
```

```
inherit         ghost ANY end
```

The module `predicate` uses the module `any` and since the module `any` uses the module `boolean` the module `predicate` uses the module `boolean` implicitly.

For any type `G` inheriting `ANY` we can form the type `PREDICATE[G]`. Predicates are used frequently in Albatross. Therefore the shorthand `G?` has been introduced to be equivalent to `PREDICATE[G]`.

The expression `p(x)` states that `x` satisfies the predicate `p`. The expression `x in p` is equivalent.

The function `<=` defines the subset relation. A predicate/set `p` is a subset of `q` if all elements of `p` occur in `q` as well.

The function `<=` has to be declared as a ghost function because it uses a universally quantified expression in its definition.

Two sets are equal if they are mutually subsets. The equality function has to be declared as ghost function as well because it uses another ghost function in its definition.

The reflexivity of equality has been proved in the implementation file, therefore the class `PREDICATE` is entitled to inherit from the class `ANY`.

The inheritance relation has to be declared as ghost inheritance because the equality function is present in the class `PREDICATE` only as a ghost function. As a consequence all functions inherited from the class `ANY` are inherited as ghost functions as well.

```
-- more declarations of the file 'predicate.ali'
0: G? = {x: false}    -- empty set
1: G? = {x: true}     -- universal set

singleton (a:G): G? -> {x: x = a}  -- singleton set with one element
     -- {a} is a shorthand for a.singleton

-- set union, intersection and difference
(+)   (p,q:G?): G?  -> {x: p(x) or q(x)}
(*)   (p,q:G?): G?  -> {x: p(x) and q(x)}
(-)   (p,q:G?): G?  -> {x: p(x) and not q(x)}

-- set complement
(-)   (p:G?): G?    -> {x: not p(x)}

-- union and intersection of a collection of sets
(+)   (pp:G??): ghost G?  -> {x: some(p) pp(p) and p(x)}
(*)   (pp:G??): ghost G?  -> {x: all(p)  pp(p) ==> p(x)}

-- Leibniz rule
all(a,b:G, p:G?)
    ensure
        a = b  ==>   p(a) ==>  p(b)
    end
```

The empty and the universal set over a type are declared as constants. Constants are defined by the equality operator and not by the arrow operator.

Albatross requires type annotations only at the top level and if the type cannot be inferred from the context. The type of the bound variable `x` in

`{x: true}` can be inferred from the context because the declared constant has type `G?` therefore the variable `x` has to have the type `G`.

An expression of the form `{x: exp}` is a predicate expression and the variable `x` can occur in the expression `exp`. This predicate describes the set of all objects `x` which satisfy the expression `exp`. The Albatross proof engine does beta reductions of the form

```
a in {x: exp_x}    -- or equivalent {x: exp_x}(a)

-- beta reduces to
exp_a   -- all variables 'x' in 'exp_x' replaced by 'a'
```

in both forwards and backwards directions.

The function `singleton` is the first non-operator function used in this tutorial. It has one argument. The most natural way to call this function is like `singleton(a)`. This form is called the functional notation. However Albatross has an equivalent object oriented notation with a dot notation, extracting the first argument and placing it as the target object, before the dot. Therefore the expression `a.singleton` is an equivalent function call.

In order to support a more mathematical notation the abbreviation `{a}` has been introduced. The language parser converts every expression of the form `{a,b,...}` to `a.singleton + b.singleton + ....`

The following functions demonstrate that function names or operators do not need to be unique in Albatross. Only the function name (operator name) together with the signature has to be unique. Therefore - can be used as a binary operator for set difference and as a unary operator for set complement.

The last declaration is the so called *Leibniz rule* of equality. The Leibniz rule expresses a concept which is very important in Albatross. If two objects are equal there is no possibility to distinguish them. The Albatross compiler enforces this rule by not allowing any definition of equality which is not strong enough to satisfy the Leibniz rule.

The Leibniz rule allows you to do rewritings. As soon as there is an equality of the form `a = b` with specific `a` and `b` in the context the proof engine does a partial specialization of the Leibniz rule and adds

```
all(p) p(a) ==> p(b)
```

to the context. How can you trigger rewritings using this rule? If you can prove the assertion `exp_a` in which `a` occurs as a subexpression you can replace this subexpression by a fresh variable `x` to get the expression `exp_x`. Now you can form the predicate `{x: exp_x}` and the subexpression `a` certainly satisfies this predicate, because you can prove `exp_a`. Now it is sufficient to state

```
b in {x: exp_x}
```

in a proof because the compiler recognizes `all(p) p(a) ==> p(b)` as a valid backward rule and tries to prove `a in {x: exp_x}` which succeeds. After the proof the proof engine forward closes the assertion `b in {x: exp_x}` by beta reduction and adds `exp_b` to the context which is `exp_a` where the subexpression `a` has been substituted by the subexpression `b`

3.4 Predicate Logic

In this chapter we proof some basic facts of predicate logic using only the modules `boolean_logic` and `predicate` of the base library. Note that the modules `boolean` and `any` of the base library are used implicitely.

```
-- file: predicate_logic_playground.al

use
    alba.base.boolean_logic
    alba.base.predicate
end
```

Many of the proofs below can be found in the module `alba.base.predicate_logic`. We use these proofs to demonstrate some important concepts of the Albatross language and its proof engine.

3.4.1 Some Simple Theorems

Recall from the chapter 3.3.2 that `0` represents the empty set or the unsatisfiable predicate and `1` represents the universal set. Evidently the empty set is a subset of any other set and the universal set is a superset of any other set. I.e. we want to be able to prove

```
0 <= p

p <= 1
```

for any predicate `p`. If we enter the assertion into our playground module we get an assertions which is proved by the proof engine automatically.

```
all(p:G?)
    ensure
        0 <= p
        p <= 1
    end
```

What happens under the hood? Let us look at the first assertion and display what the proof engine generates.

```
all(p:G?)
    proof -- generated by the proof engine
        all(x)   -- entering and deduction rule
            require
                x in 0
            proof
                x in {x: false}      -- '0' expanded
                false                -- beta reduction
                all(a) a             -- ex falso quodlibet
            ensure
                x in p               -- trivially valid
            end

        all(x) x in 0  ==>  x in p   -- '<=' expanded
    ensure
        0 <= p
    end
```

We have to read this proof first bottoms up. The proof engine looks at the target `0 <= p` and expands the definition. The new goal is a universally quantified expression. This is proved by entering it. Entering a universally quantified goal means to generate a new context within which all bound variables are visible. The internal goal of the universally quantified expression is an implication, therefore the deduction rule can be applied shifting the antecedent into the context.

Now the proof engine can apply forward reasoning to the assumption `x in 0`. The `in` operator cannot be expanded, because no definition is available. But the right operand `0` has a defintion which can be unfolded resulting in `x in {x:false}`. Then beta reduction can be applied to this expression putting `false` into the context. Now the proof engine specializes the law *ex falso quodlibet* from the module `boolean_logic` putting `all(a) a` into the context. Having this all goals can be proved immediately.

At this point it might be interesting to understand the precise rules which govern function expansion and beta reduction.

1. No functions are expanded in universally quantified expressions, implications and conjunctions.

2. If the expression has at the toplevel a function whose definition is available it expands the definition (the function `<=` in the above example).

3. If the expression can be beta reduces at the toplevel then beta reduction is done.

4. If the expression has at the top a function with no definition available all definitions below are expanded recursively.

The expression `x in 0` is expanded to `x in {x:false}` because no definition is available for the `in` operator and the constant `0` has a definition. If there were available a definition for the `in` operator then the subterm `0` would not be expanded.

The same applies to beta reductions. If a beta reduction is possible at the top then no definitions nor beta reductions in inner subterms are applied.

Exercise: Explain how the proof engine proves the assertions

```
p <= 1
p * (-p) = 0
p + (-p) = 1
```

3.4.2 Singleton Sets

A singleton set is a set with one element e.g. `{a}`. Note that `{a}` is just a shorthand for `singleton(a)` which has the definition `{x: x=a}`.

A singleton set has one element and therefore cannot be empty. We should be able to prove

```
{a} /= 0
```

This goal is expanded to `{a} = 0 ==> false`, the antecedent is shifted into the context and `false` is used as a new goal. With the antecedent in the context the proof engine starts forward resoning by entering the following expressions into the context:

```
{a} = 0
{a} <= 0
0   <= {a}
all(x) x in {a} ==> x in 0
all(x) x in 0   ==> x in {a}
```

Now no rule can be applied for backward reasoning and the proof engine cannot continue to prove the goal `false`.

We need to give the proof engine a hint on how to continue. If we look the result of forward reasoning we discover that `x in 0` for any `x` would do the job. `0` could be expanded and betareduction would resolve the goal. I.e. we need some `x` which is in `{a}`. Clearly `a` is in `{a}`. Now we have the missing link and can formulate a complete proof.

```
all(a:G)
    proof
        a in 0  ==>  false
    ensure
        {a} /= 0
    end
```

3.4.3 Leibniz Equality and Term Rewriting

The module `predicate` of the base library asserts the Leibniz rule of equality.

```
-- from the module 'predicate'
all(a,b:G, p:G?)
    ensure
        a = b  ==>  p(a)  ==>  p(b)
    end
```

This rule states that equal objects cannot be distinguished by any property, i.e. if `a` has a certain property then `b` has it as well.

As already explained in the chapter 3.3.2 we can use the Leibniz rule to do term rewritings. Here we use the Leibniz rule to prove the symmetry and transitivity of equality i.e. we want to prove the laws

```
a = b  ==>  b = a                -- symmetric

a = b  ==>  b = c  ==>  a = c    -- transitive
```

Let's look first at symmetry. The proof engine applies the deduction rule and shifts `a = b` into the context. Having this it can partially specialize the Leibniz rule to get `all(p) p(a) ==> p(b)` into the context. No more forward nor backward steps are possible and the proof engine cannot continue.

The goal is `b = a`. If we want to explore the Leibniz rule we have to express this a property applied to `b`. The expression `b in {a}` will do the job because this expression expanded yields the goal `b = a`. The proof engine can proof `b in {a}` by using the partially specialized Leibniz rule as a backward rule.

```
all(a,b:G)
    require
        a = b
    proof
        b in {a}
    ensure
        b = a
    end
```

In proving transitivity the proof engine shifts the two antecedents into the context and forward closes it by doing partial specialization of the Leibniz rule. The context for the goal `a = c` looks like

```
a = b
b = c
all(p) p(a) ==> p(b)
all(p) p(b) ==> p(c)
```

Because we have already proved the symmetry of equality the proof engine can do some backward reasoning and try to prove `c = a` instead of `a = c`. We can express the latter by the equivalent expression `c in {a}`. This expression can be proved by the engine by using the partially specialized Leibniz rules and doing backward reasoning reaching at `a in {a}` which is proved trivially.

```
all(a,b,c:G)
    require
        a = b
        b = c
    proof
        c in {a}
    ensure
        a = c
    end
```

Having proved the transitivity of an operator the proof engine is able to follow transitivity chains. The following proof discharges automatically.

```
all(a,b,c,d,e:G)
    require
        a = b
        b = c
        c = d
        d = e
    ensure
        a = e
    end
```

3.4.4 Existential Quantification

An existential quantified expression in Albatross looks like

```
some(x,y,...)  expression
```

where the expression contains the variables `x,y,...`. The proof engine of Albatross has two hardwired procedures to handle existentially quantified expressions.

Introduction rule: If the goal of a proof is an existentially quantified expression it looks for witnesses i.e. it tries to find expressions in the context with specific values for the variables so that the expression with the specific values is valid. If there are witnesses then the existentially quantified expression is proved.

Elimination rule: Whenever an existentially quantified expression is entered into the context, the proof engine enters the following assertions into the context.

```
all(c) (all(x,y,...) expression ==> c) ==> c
```

The introduction rule is immediately evident. The elimination rule mimics the following mathematical proof technique:

> We know that there is some x which satisfies p. Let a be an arbitrary value satisfying p. From a satisfying p we conclude c. Therefore c is valid.

We use these laws to prove that every set which has elements is nonempty and every nonempty set has elements i.e.

```
some(x) x in p  ==>  p /= 0

p /= 0  ==>  some(x) x in p
```

For the first one the proof engine ends up in the following context/goal pair and then gets stuck.

```
some(x) x in p

all(c) (all(x) x in p ==> c) ==> c

p = 0                          -- (1)

p <= 0

all(x) x in p  ==> x in 0      -- (2)
...
================================
false
```

Now we have to give the proof engine a hint how to substitute `c` so that `all(x) x in p ==> c` can be proved in the context and `c` implies a contradiction within the context. We note that we can substitute anything for `c` because (2) guarantees that with `x in p` in the context we get an immediate contradiction and can prove anything. A good choice for `c` is `p /= 0` because together with (1) we get the needed contradiction to prove `false`. The complete proof:

```
all(p:G?)
    require
        some(x) x in p
    proof
        all(x) x in p ==> p /= 0
    ensure
        p /= 0
    end
```

Now the other direction `p /= 0 ==> some(x) x in p`. With this goal the proof engine becomes stuck very fast. It just enriches the context by expanding `/=` and then `not` and finds nothing more to continue. We cannot find any way to generate a witness for the existentially quantified expression. If all else fail we can try a proof by contradiction i.e. we add the negated goal to the context and try to derive `false` from it. Lets see the steps the proof engine would do.

```
all(p:G?)
    require
        p /= 0
        not some(x) x in p
    proof
        p = 0  ==>  false
        (some(x) x in p)  ==> false

        all(x)
            require
                x in p
            proof
                some(x) x in p
                false
            ensure
                x in 0
            end
        p <= 0
        p = 0
    ensure
        false
    end
```

First it does forward reasoning on the two assumptions and gets two implications into the context which have `false` as a target. Then uses the first in a backward reasoning step with the new goal `p = 0`. For this goal in has to prove `all(x) x in p ==> x in 0`. It enters it and realizes that it needs a further contradiction to prove it. Then it uses the second implication and gets the new goal `some(x) x in p`. This goal can be resolved immediately because there is a witness in the context.

The only thing needed by the proof engine is a trigger to start a proof by contradiction. Therefore the following proof is complete.

```
all(p:G?)
    require
        p /= 0
    proof
        not not some(x) x in p
    ensure
        some(x) x in p
    end
```

3.5 Using and Proving Abstractions

In this chapter we show how to use deferred classes to create abstractions. Abstractions are very useful because they allow us to define functions based

on some deferred functions and to prove assertions based on some deferred assertions.

The inheritance mechanism of Albatross is a powerful reuse mechanism. All classes which inherit from an abstraction get all functions defined in the abstraction and all assertions proved in the abstraction for free.

We demonstrate the concepts by creating an abstraction for partial orders and linear orders. The base library already has these abstractions. In this chapter we create some simplified versions of the abstractions in the base library. After having read this chapter it might be interesting to read the versions in the base library.

3.5.1 Partial Order

In mathematics a partial order is a set with a binary relation which is reflexive, antisymmetric and transitive.

It is easy to model such an abstraction in Albatross.

```
-- file: partial_order.al
use alba.base.predicate end

deferred class PARTIAL_ORDER end

PO: PARTIAL_ORDER

(=)  (a,b: PO): BOOLEAN  deferred end
(<=) (a,b: PO): BOOLEAN  deferred end
(<)  (a,b: PO): BOOLEAN  -> a <= b and a /= b
(>=) (a,b: PO): BOOLEAN  -> b <= a
(>)  (a,b: PO): BOOLEAN  -> b < a

all(a,b,c:PO)
    deferred ensure
        a = a
        a <= a                                  -- reflexivity
        a <= b  ==>  b <= a  ==>  a = b         -- antisymmetry
        a <= b  ==>  b <= c  ==>  a <= c        -- transitivity
    end

deferred class PARTIAL_ORDER inherit ANY end
```

The transitivity of the >= operator is trivial to prove

```
all(a,b,c:PO)
    require
        a >= b
        b >= c
    ensure
        a >= c
    end
```

Let's understand why this proof succeeds. After shifting the assumptions into the context the proof engine forward closes the context. Forward closing includes expansion of functions. Therefore after shifting the assumptions the context consists of

```
a >= b;  b >= c              -- assumptions

b <= a                       -- function expansion

c <= b                       -- function expansion

all(x) b <= x  ==> c <= x    -- partial specialization of transitivity

...
```

Now the proof engine tries to prove the goal. It cannot prove it immediately. Therefore it searches for and generates backward rules. Generation of backward rule includes function expansion. The expanded goal is `c <= a`. The expanded goal matches the conclusion of the partial specialization and therefore backward reasoning can continue. The substituted premise is already in the context and therefore the proof succeeds.

This little proof is a nice demonstration how forward and backward reasoning work hand in hand and that the proof engine can do many detailed steps which would be very tedious to do them by hand.

The `<` operator should be transitive as well. However the proof engine fails to prove the transitivity without help.

```
all(a,b,c:PO)
    require
        a < b;  b < c
    ensure
        a < c        -- PROOF FAILS!
    end
```

Let's see the result of forward reasoning after shifting the assumptions

```
a <= b
a /= b
a = b ==> false
b <= c
b /= c
b = c ==> false
...
```

Backward reasoning generates the following subgoals

```
require  a = c
proof    ...
ensure   false end

a = c ==> false
a /= c
a <= c  -- proved by transitivity of '<='
a < c
```

I.e. the proof can succeed if the assumption of `a = c` leads to falsehood. In order to prove falsehood the proof engine generates the subgoals

```
...
b <= c
c <= b
b = c
false
```

Now it might be clear what the missing link is. The proof engine cannot prove `c <= b`. However we have `a <= b` and `a = c` in the context. If we trigger a term rewrite of the first by the second one we get `c <= b` into the context and the proof succeeds. Term rewriting can be triggered by applying the Leibniz rule (see chapter on the module `predicate`).

The following successfully proves the transitivity of `<`.

```
all(a,b,c:PO)
    require
        a < b
        b < c
    proof
        require
            a = c
        proof
            c in {x: x <= b}
        ensure
            false
        end
    ensure
        a < c
    end
```

Note that the proof engine needs just a little hint on how to conclude falsehood from the assumption `a = c`. It is sufficient to provide this hint. The detailed steps, which are numerous, are filled in by the proof engine.

In the base library we have already an example of a class which satisfies the requirements of a partial order. The subset relation defined in the module `predicate` is certainly reflexive, antisymmetric and transitive.

We can prove this conjecture by

```
G: ANY

all(p,q,r:G?)
    ensure
        p <= p

        p <= q  ==>  q <= p  ==>  p = q

        p <= q  ==>  q <= r  ==>  p <= q
    end
```

By using function expansion and applying the basic techniques already seen in this tutorial it is evident that the proof engine proves these claims successfully.

Having reflexivity, antisymmetry and transitivity of the subset relation we can state that the class `PREDICATE` is entitled to inherit the class `PARTIAL_ORDER`.

```
immutable class
    predicate.PREDICATE[G]
inherit
    ghost PARTIAL_ORDER
end
```

Now the class `PREDICATE` inherits all functions defined in this module (`<`, `>=`, `<`) and all assertions proved in this module.

But in order to see that the abstraction is really useful we need more *clients* which use it. In the next chapter we will see another class which can inherit from a partial order.

But before using the module we have to create an interface file for it. We don't show the interface file here because looking at the interface files already shown here it should be clear what it has to contain.

3.5.2 Linear Order

A linear order is the special case of a partial order where all elements are comparable.

```
-- file: linear_order.al
deferred class LINEAR_ORDER end

LO: LINEAR_ORDER

(=)  (a,b:LO): BOOLEAN  deferred end
(<=) (a,b:LO): BOOLEAN  deferred end

all(a,b,c:LO)
    deferred ensure
        a = a

        a <= b  or  b <= a

        a <= b  ==>  b <= a  ==>  a = b

        a <= b  ==>  b <= c  ==>  a <= c
    end

deferred class LINEAR_ORDER inherit ANY end
```

Note that we have not included reflexivity because reflexivity is a consequence of these deferred assertions.

```
all(a:LO)
    proof
        a <= a  or  a <= a
    ensure
        a <= a
    end
```

Now we have enough evidence to state that a linear order is a special case of a partial order which is stated as an inheritance relation in Albatross

```
deferred class
    LINEAR_ORDER
inherit
    PARTIAL_ORDER
end
```

In a partial order pairs of elements can be unrelated. In a linear order this cannot happen. This is the content of the second deferred assertion.

Therefore if we know that `not (a <= b)` holds we must be able to conclude `b < a`. In order to prove this conclusion we have to prove `b <= a` and `b /= a`.

The first one is a consequence of the fact that pairs are always related by `<=` and if one relation does not hold then the other must hold. The second one can be proved by showing the assumption `b = a` leads to a contradiction.

```
all(a,b:LO)
    require
        not (a <= b)
    proof
        a <= b  or  b <= a

        require  b = a
        proof    a in {x: x <= b}
        ensure   false end
    ensure
        b < a
    end
```

Note that we used the Leibniz rule to rewrite the assertion `b <= b` to `a <= b` which contradicts the assumption.

We want to prove as well that `not (a < b)` leads to `b <= a`. We use a proof by contradiction i.e. we assume that the conclusion does not hold. From the last assertion we know that this leads to `a < b` which contradicts the assumption.

```
all(a,b:LO)
    require
        not (a < b)
    proof
        require  not (b <= a)
        proof    a < b
        ensure   false end
    ensure
        b <= a
    end
```

In a linear order two different elements have to be related in the strict order relation `<`.

```
all(a,b:LO)
    require
        a /= b
    proof
        require
            not (a < b)    -- implies b <= a
        ensure
            b < a
        end
    ensure
        a < b  or  b < a
    end
```

3.6 Inductive Data Types

Using inductive data types is a very powerful way to define the structure of data. Functional languages or languages with a strong functional component

like Ocaml, Haskell, FSharp, Swift, Coq, etc. use inductive data types as its central concept to define data. Inductive data types are also known as algebraic data types.

3.6.1 Basics of Inductive Types

A very simple inductive type is an enumeration type. To define a type which represents a day of the week we can use the declaration

```
case class
    DAY
create
    monday
    tuesday
    wednesday
    thursday
    friday
    saturday
    sunday
end
```

The keyword `case` tells the compiler that an inductive type is being declared. Each inductive type has a list of constructors which consists at least of one element. The above type has one constructor for each day of the week.

Pattern matching can be used to define a function which maps each weekday to its next weekday.

```
next (d:DAY): DAY
    -> inspect d
       case monday    then tuesday
       case tuesday   then wednesday
       case wednesday then thursday
       case thursday  then friday
       case friday    then saturday
       case saturday  then sunday
       case sunday    then monday
       end
```

The Albatross compiler can use pattern matching within proofs to do term rewriting. E.g. it can rewrite the term `next(monday)` to `tuesday`. We can use this feature to prove the following assertions:

```
all ensure
    next(sunday) = monday

    sunday.next.next.next = wednesday
end
```

Note that in Albatross the terms `f(x)` and `x.f` are equivalent. The object oriented notation is more convenient for iterated function applications because nested parentheses can be avoided.

The compiler generates for each inductive type an equality function. Therefore the expressions in the previous proof are well typed. Furthermore each inductive type automatically inherits ANY.

The compiler guarantees that two objects created with different constructors are different. I.e. the following assertions are generated by the compiler.

```
all ensure
        -- generated by the compiler
    monday  = tuesday    ==>  false
    tuesday = wednesday  ==>  false
    ... -- all other possible combinations
end
```

These inversion laws can help us to do proofs by contradiction.

Inductive types can contain data. An inductive type can be declared which has an optional element and an inductive type can be declared which has a list of elements.

```
G: ANY

case class
    OPTION[G]
create
    none
    value (item:G)
end

case class
    LIST[G]
create
    []
    (^) (head:G, tail:LIST[G])
end
```

The compiler generates automatically deconstructors for inductive types with data. For the above types the following functions are generated.

```
item (o:OPTION[G]): G          -- generated by the compiler
    require
        o as value(_)
    ensure
        Result = inspect o
                 case value(v) then v
                 end
    end

head (l:LIST[G]): G            -- generated by the compiler
    require
        l as _^_               -- '_' is an unnamed variable
    ensure
        Result = inspect l
                 case h ^ _  then h
                 end
    end

tail (l:LIST[G]): LIST[G]      -- generated by the compiler
    require
        l as _^_
    ensure
        Result = inspect l
                 case _ ^ t  then t
```

```
                end
    end
```

The boolean expression `o as value(_)` tests if the object `o` has been constructed with the constructor `value`. The underline character _ is used for variables whose name is not interesting.

Clearly all these functions need preconditions because the used pattern matching in the definition of the functions is not complete. You cannot extract an element if the optional has been constructed with `none` and you cannot get neither the head nor the tail of an empty list.

3.6.2 Recursion and Induction

Natural numbers can be declared with an inductive type.

```
case class
    NATURAL
create
    0
    successor(predecessor:NATURAL)
end
```

This definition says that a natural number is either `0` or the successor of some other natural number which is called its predecessor. There are no other possibilities to construct natural numbers.

The addition of two natural numbers can be defined recursively.

```
(+) (a,b:NATURAL): NATURAL
    -> inspect b
       case 0 then a
       case successor(n) then (a + n).successor
       end
```

Albatross allows only recursive calls which are sound i.e. whose termination is guaranteed. Therefore the recursion checker of the Albatross compiler checks two things for a recursive call:

- At least one argument of the recursive call has to be structurally smaller than the original argument.
- The recursive call has to be within a branch.

The above definition of the function `+` satisfies both criteria. The argument `n` is the predecessor of the argument `b` of the original call and the call occurs within a case branch of an inspect expression.

The first criterion is needed to guarantee that the execution of the function at runtime terminates. The second one is needed to guarantee that unfolding definitions at compile time is guaranteed to terminate because the compiler unfolds definitions outside of branches arbitrarily and within branches only if it can prove that the branch is entered.

In order to prove properties of recursive functions we need an induction law. The compiler generates for all inductive types an induction law. Examples:

```
all(p:DAY?)   -- generated by the compiler
    require
        p(monday);   p(tuesday); p(wednesday)
        p(thursday); p(friday)
        p(saturday); p(sunday)
    ensure
        all(d) d in p
    end
```

In order to prove that all days of the week satisfy a certain property we have to prove that `monday` satisfies the property, `tuesday` satisfies the property, ..., and `sunday` satisfies the property.

```
all(p:OPTION[G]?)   -- generated by the compiler
    require
        none in p
        all(v) value(v) in p
    ensure
        all(o) o in p
    end
```

In order to prove that all objects with an optional element satisfy a certain property we have to prove that `none` satisfies the property and that for all values `v` the object `value(v)` satisfies the property.

We see that for each constructor we get a premise to satisfy in order to prove that all objects satisfy the property.

Now let us look at lists which have one recursive constructor (`^`) which needs an already constructed list to construct a new list by prepending a head element in front of the list. For lists we get the induction principle

```
all(p:LIST[G]?)   -- generated by the compiler
    require
        []  in p
        all(h,a) a in p ==> h^a in p
    ensure
        all(a) a in p
    end
```

We see that each constructor gets a premise to satisfy. However the recursive constructor gets a premise with an induction hypothesis.

Let us express the induction law in words in order to understand it.

In order to prove that all lists satisfy a certain property we have to prove that the empty list satisfies the property and for all elements and lists it has to be shown: If the list satisfies the property then the list with the new element in front satisfies the property as well.

Since all lists have to be constructed with one of the constructors only list which satisfy the property can be constructed.

The generated induction principle for natural numbers is very similar to the one generated for lists.

```
all(p:NATURAL?)   -- generated by the compiler
    require
        0 in p
        all(n) n in p ==> n.successor in p
```

```
    ensure
        all(n) n in p
    end
```

Since the constructor `successor` is recursive (i.e. it needs a natural number to construct a new one) the premise which corresponds to this constructor gets an induction hypothesis as well.

There remains one little detail. The proof engine of Albatross bubbles up all universal quantifiers which appear in the conclusion of a law i.e. for natural numbers the compiler generates the induction principle in the form

```
all(p:NATURAL?, n:NATURAL)    -- generated by the compiler
    require
        0 in p
        all(n) n in p ==> n.successor in p
    ensure
        n in p
    end
```

which is equivalent to the former presentation (Note that the variable `n` does not occur free in the premises). In this form the proof engine can work with it more systemtically.

Now let's see how to prove some properties of addition by using the induction law of natural numbers.

If the declared type `NATURAL` correponds to our intuitive notion of natural numbers we should be able to prove `0 + a = a` for all natural numbers `a`. We generate the proof step by step in all details.

In order to use the induction principle we need a property i.e. a predicate over natural numbers. Since there is only one variable there is no choice.

```
a in {a: 0 + a = a}
```

This expression if obviously equivalent to our goal. But in this form it matches the conclusion of the induction law. The template of the proof by induction looks like

```
all(a:NATURAL)
    proof
        ...
        0 in {a:NATURAL: 0 + a = a}  -- explicit type to avoid
                                     -- ambiguities
        all(n:NATURAL)
            require
                n in {a: 0 + a = a}
            proof
                0 + n = n
                ...
                0 + n.successor = n.successor
            ensure
                n.successor in {a: 0 + a = a}
            end

        a in {a: 0 + a = a}
    ensure
        0 + a = a
    end
```

The induction start is verified by the proof engine by making the following backward steps:

```
-- read bottoms up
0 = 0                          -- reflexivity of '='

0 + 0 = 0                      -- apply function '+'

0 in {a:NATURAL: 0 + a = a}    -- substitute 'a' by '0'
```

The induction step is verified by backward reasoning as well.

```
-- read bottoms up
n.successor = n.successor        -- reflexivity of '='

(0 + n).successor = n.succesor   -- use induction hypothesis

0 + n.successor = n.successor    -- apply function '+'

n.successor in {a: 0 + a = a}    -- substitute 'a' by n.successor
```

Note that nearly all steps presented for this proof are done automatically by the proof engine. The proof engine needs just a hint to do a proof by induction. Therefore the following minimal version of the proof is sufficient.

```
all(a:NATURAL)
    proof
        0 in {a:NATURAL: 0 + a = a}
        a in {a:NATURAL: 0 + a = a}
    ensure
        0 + a = a
    end
```

This is the standard form of a proof by induction where the proof engine receives just a trigger to apply the induction law and can do the rest of the work alone. The first assertion is proved trivially and after the proof forward reasoning puts the following partial specialization of the induction law into the context

```
all(n)
    (all(n) n in {a:NATURAL: 0 + a = a}
            ==> n.successor in {a:NATURAL: 0 + a = a})
    ==>
    n in {a:NATURAL: 0 + a = a}
```

The second assertion matches the target of this partial specialization and can therefore trigger backward reasoning which succeeds automatically as demonstrated above.

Let's come to the next law of addition: associativity. In order to prove it the proof engine needs just a hint to apply the induction law.

```
all(a,b,c:NATURAL)
    proof
        0 in {c: (a + b) + c = a + (b + c)}
        c in {c: (a + b) + c = a + (b + c)}
    ensure
        (a + b) + c = a + (b + c)
    end
```

It might be interesting for the reader to prove this assertion in the same manner as the proof engine.

The proof of commutativity of addition (i.e. `a + b = b + a`) is little bit more complex. Again we trigger the induction law to get the partially finished proof:

```
all(a,b:NATURAL)
    proof
        ...
        0 in {b: a + b = b + a}

        all(b)
            require
                b in {b: a + b = b + a}
            proof
                a + b = b + a
                ...
                (a + b).successor = b.successor + a
                a + b.successor = b.succesor + a
            ensure
                b.successor in {b: a + b = b + a}
            end

        b in {b: a + b = b + a}
    ensure
        a + b = b + a
    end
```

Now look at the induction start. The proof engine tries backward reasoning

```
-- read bottoms up
a = 0 + a                    -- proof engine cannot continue!!

a + 0 = 0 + a                -- apply '+' to the left hand side

0 in {b: a + b = b + a}      -- substitute 'b'
```

Why is the proof engine stuck here? We have already proved the law `0 + a = a` and from the module `predicate_logic` we know that `=` is symmetric. The problem: The above presented backward steps are not completely correct. When the proof engine sees `a + 0 = 0 + a` it unfolds definitions until no more unfolding is possible. Therefore we do not get `a = 0 + a` as a goal because the addition can still be unfolded. Instead we get

```
a = inspect a
    case 0 then 0
    case successor(n) then (0 + n).successor
    end
```

Unfortunately this goal cannot be resolved. We have to tell the proof engine to use `0 + a = a` as a simplification. We do this by writing

```
proof  0 + a = a
ensure 0 in {b: a + b = b + a} end
```

Now the proof engine can do the following backward reasoning steps:

```
-- read bottoms up
a = a                        -- proved by reflexivity of '='
```

```
a + 0 = a                    -- apply '+' to the left hand side

a + 0 = 0 + a                -- apply simplification

0 in {b: a + b = b + a}      -- substitute 'b'
```

Now we have to look at the unfinished induction step.

```
  a + b = b + a
  ...
  (a + b).successor = b.successor + a
```

The first term can be transformed via the induction hypothesis into

```
  (a + b).successor = (b + a).successor   -- induction hypothesis
  (b + a).successor = b + a.successor     -- apply '+' to right hand side
```

Now the remaining missing link is

```
  ...
  b + a.successor = b.successor + a
```

Unfortunately this statement cannot be proved directly it needs an own proof by induction. If we try induction on `a` we get the induction step

```
  b + a.successor = b.successor + a
  ...
  (b + a.successor).successor = (b.successor + a).successor
  b + a.successor.successor   = b.successor + a.successor
```

which succeeds trivially by applying the induction hypothesis. It is good practive to separate out this proof.

```
all(a,b:NATURAL)
        -- lemma
    proof
        0 in {a: b + a.successor = b.successor + a}
        a in {a: b + a.successor = b.successor + a}
    ensure
        b + a.successor = b.successor + a
    end
```

Having this lemma the proof of commutativity of addition can be completed.

```
all(a,b:NATURAL)
    proof
        proof  0 + a = a
        ensure 0 in {b: a + b = b + a} end

        all(b)
            require
                a + b = b + a
            proof
                a + b.successor   = (a + b).successor  -- apply '+'
                (a + b).successor = (b + a).successor  -- induction
                                                       --  hypothesis
                (b + a).successor = b + a.successor    -- apply '+'
                b + a.successor   = b.successor + a    -- lemma
            ensure
```

```
                a + b.successor = b.successor + a
            end

        b in {b: a + b = b + a}
    ensure
        a + b = b + a
    end
```

In this proof just the essential steps have been kept and nearly all steps which can be done by the proof engine automatically have been left out. This has the advantage that we not only have a proof which can be formally checked but also a proof which presents all important steps to the reader.

3.6.3 Inversion and Injection Laws

Work in progress.

3.6.4 Binary Tree

A binary tree is either empty (i.e. a leaf) or it contains an information element and a left and a right subtree. A corresponding declaration in Albatross formalizes this informal description.

```
case class
    BINARY_TREE[G]
create
    leaf
    tree (info:G, left,right:BINARY_TREE[G])
end
```

The generated induction principle is

```
all(p:BINARY_TREE[G]?, t:BINARY_TREE[G])
    require
        leaf in p

        all(i,l,r) l in p ==> r in p ==> tree(i,l,r) in p
    ensure
        t in p
    end
```

We write a function to mirror a binary tree as a recursive function

```
(-) (t:BINARY_TREE[G]): BINARY_TREE[G]
    -> inspect t
       case leaf        then leaf
       case tree(i,l,r) then tree(i,-r,-l)
       end
```

The mirroring of a tree should be an involution i.e. mirroring a tree twice should return the original tree. This assertion can be proved by induction.

```
all(t: BINARY_TREE[G])
    proof
        leaf in {t: t = - (- t)}
```

```
        all(i,l,r)
            require
                l = - (- l)
                r = - (- r)
            ensure
                tree(i,l,r) = - (- tree(i,l,r))
            end

        t in {t: t = - (- t)}
    ensure
        t = - (- t)
    end
```

We have used three intermediate assertions to prove the goal. For the proof engine the second is unnecessary. It has been added here for documentation purposes.

As a next step we define a recursive function which calculates the inorder sequence of a tree.

```
-- note: for the following we assume that the module 'list' of the
--       base library is used in the current module!

inorder (t:BINARY_TREE[G]): [G]
    -> inspect t
       case leaf then []
       case tree(i,l,r) then l.inorder + i ^ r.inorder
       end
```

In words: The inorder sequence of an empty tree is the empty list. The inorder sequence of a nonempty tree is the inorder sequence of the left subtree concatenated with info-prefixed inorder sequence of the right subtree.

Note that we use the module `list` of the base library. It has the same declaration as the list of the previous chapters. However since lists are used very frequently the type `LIST[G]` of the basic libary can be abbreviated with `[G]`.

Furthermore the parser of the Albatross language converts every expression of the form `[a,b,...z]` to `a^b^...^z^[]` . This mechanism is a pure macro facility i.e. it works for any type which declares a constant `[]` and a binary operator `^` (provided that the expanded expression passes the type checker).

The module `list` of the base library provides a lot of functions and proves a lot of useful assertions. In the following we use the following functions and assertions.

```
-- from the module 'alba.base.list'

(+) (a,b:[G]): [G]
        -- The concatenation of the lists 'a' and 'b'
    -> inspect a
       case []  then b
       case h^t then h^(t + b)
       end

(-) (a:[G]): [G]
        -- The mirrored list 'a'
```

```
    -> inspect a
       case []  then []
       case h^t then - t + [h]
       end

all(a,b,c:[G])
    ensure
        (a + b) + c = a + (b + c)    -- associative

        (- (a + b)) = -b + -a        -- mirror concatenation

        a = - (- a)                  -- involution
    end
```

What is the inorder sequence of a mirrored tree? Obviously the mirrored inorder sequence of the original tree. The following assertion proves this claim.

```
all(t:BINARY_TREE[G])
proof
    leaf in {t: (-t).inorder = - t.inorder}

    all(i:G, l,r:BINARY_TREE[G])
        require
            (-l).inorder = - l.inorder
            (-r).inorder = - r.inorder
        proof
            (-tree(i,l,r)).inorder            -- unfold definitions
                = -r.inorder + ([i] + -l.inorder)

            (-r).inorder + ([i] + -l.inorder) -- associativity
                = ((-r).inorder + [i]) + -l.inorder

            ((-r).inorder + [i]) + -l.inorder -- unfold definition
                = -i^r.inorder + -l.inorder

            (-i^r.inorder) + -l.inorder       -- mirror concatenation
                = - (l.inorder + i ^ r.inorder)

            (- (l.inorder + i ^ r.inorder))   -- unfold definition
                = - tree(i,l,r).inorder
        ensure
            (-tree(i,l,r)).inorder = - tree(i,l,r).inorder
        end

    t    in {t: (-t).inorder = - t.inorder}
ensure
    (-t).inorder = - t.inorder
end
```

Note how a proof expressed in this form serves two purposes. (1) It can be formally verified by the proof engine. (2) It provides enough information for the human reader to convince him that the proof is correct.

3.6.5 List Sorting

In this chapter we develop a verified insertion sort algorithm for lists. For the verification of the algorithm we use some functions and theorems of the module

`list` of the base library. For completeness the used functions and theorems are listed here.

```
-- from the module 'alba.base.list'
G: ANY

size (a:[G]): NATURAL
    -> inspect a
       case []  then 0
       case h^t then t.size.successor
       end

(in) (x:G, a:[G]): BOOLEAN
        -- Is the element 'x' contained in the list 'a'?
    -> inspect a
       case []  then false
       case h^t then x=h or x in t
       end

all_in (a:[G], p:G?): BOOLEAN
        -- Do all elements of the list 'a' satisfy the predicate 'p'?
    -> inspect a
       case []  then true
       case h^t then h in p and t.all_in(p)
       end

all_in (a,b:[G]): BOOLEAN
        -- Are all elements of the list 'a' contained in the list 'b'?
    -> a.all_in(elements(b))


same_elements (a,b:[G]): BOOLEAN
        -- Have the lists 'a' and 'b' the same elements
    -> a.all_in(b) and b.all_in(a)


permutation (a,b:[G]): ghost BOOLEAN
    -> a.size = b.size and same_elements(a,b)

all(a:[G], p,q:G?)
    ensure
        x in a ==> a.all_in(p) ==> x in p
        a.all_in(b) ==> b.all_in(p) ==> a.all_in(p)
        a.all_in(p) ==> p <= q ==> a.all_in(q)

        permutation(a,a)
        permutation(a,b) ==> permutation(b,a)
        permutation(a,b) ==> permutation(b,c) ==> permutation(a,c)
        permutation(x^y^a, y^x^a)
        permutation(a,b) ==> permutation(x^a,x^b)
    end
```

The sorting code we develop in this chapter need some modules of the base library.

```
use
    alba.base.boolean_logic
    alba.base.predicate_logic
```

```
    alba.base.linear_order
    alba.base.list
end
```

In an implementation file it is usually convenient to use the modules for boolean logic and predicate logic because they provide us with many useful laws of logic. We need a linear order because only list of elements which have a linear order can be sorted. Furthermore we use the lists of the base library because we want to sort lists.

Before trying to write a search function we need to define what it means for a list to be sorted and the properties which sorted lists have. First we define a function which tells whether an element is less than or equal all elements in the list.

```
L: LINEAR_ORDER

is_lower_bound (x:L, a:[L]): BOOLEAN
    -> a.all_in({y: x <= y})
```

This lower bound function has the following transitivity property:

```
all(x,y:L, a:[L])
        -- transitivity
    require
        x <= y
        y.is_lower_bound(a)
    proof
        {z: y <= z} <= {z: x <= z}
    ensure
        x.is_lower_bound(a)
    end
```

Furthermore we expect that a lower bound for a list is a lower bound for any permutation of the list.

```
all(x:L, a,b:[L])
    require
        x.is_lower_bound(a); permutation(a,b)
    ensure
        x.is_lower_bound(b)
    end
```

Fortunately the proof engine accepts this assertion without any intervention. It just expands the definitions of lower bounds and permutations.

Now we have to define the notion of a sorted list. The empty list and the one element list are always sorted. A list having more than one element is sorted if the first two elements are sorted and the tail of the list is sorted.

```
is_sorted (l:[L]): BOOLEAN
    -> inspect l
       case []  then true   -- empty list
       case x^t then
           inspect t
           case []  then true  -- one element list
           case y^a then x <= y and t.is_sorted
           end
       end
```

Note: Nested inspect expressions are needed because of a restriction in the version 0.2 of the compiler. In future versions a more readable equivalent expression can be used.

```
inspect l
case []    then true
case [_]   then true
case x^y^t then x <= y and (y^t).is_sorted
end
```

Now let us think a moment about properties which sorted lists should have.

- The tail of a sorted nonempty list should be sorted.
- The head of a sorted list should be a lower bound for the tail of the list.
- If a list is sorted then any lower bound of the list prepended should result in a sorted list as well.

These three properties can be proved by induction by just telling the proof engine that we want a proof by induction.

```
all(x:L, a:[L])
    proof
        []  in {a: (x^a).is_sorted ==> a.is_sorted}
        a   in {a: (x^a).is_sorted ==> a.is_sorted}
    ensure
        (x^a).is_sorted ==> a.is_sorted
    end
```

```
all(x:L, a:[L])
    proof
        []  in {a: all(x) (x^a).is_sorted ==> x.is_lower_bound(a)}
        a   in {a: all(x) (x^a).is_sorted ==> x.is_lower_bound(a)}
    ensure
        (x^a).is_sorted ==> x.is_lower_bound(a)
    end
```

```
all(x:L, a:[L])
    require
        x.is_lower_bound(a)
        a.is_sorted
    proof
        []  in {a: x.is_lower_bound(a)
                   ==>  a.is_sorted
                   ==> (x^a).is_sorted}
        a   in {a: x.is_lower_bound(a)
                   ==>  a.is_sorted
                   ==> (x^a).is_sorted}
    ensure
        (x^a).is_sorted
    end
```

Note that in the second assertion the universal quantification over the elements `x` has been shifted into the predicate for the induction proof. It is evident

that the proof, if succeeds, implies the wanted assertion. But this stronger predicate for the induction proof results in a stronger induction hypothesis (because it is universally quantified over the head element). With the stronger induction hypothesis the proof succeeds, without it fails.

Now we have enough assertions to design a function which inserts an element into a sorted list maintaining the list sorted. Inserting into an empty list is trivial since the one element list is always sorted.

Look at the case that we want to insert an element `x` into an already sorted list `y^a`. There are two cases: `x <= y` and `not (x <= y)`. In the first case the new element is a lower bound of the list and can therefore be prepended without destroying the sorting. In the second case we know that `y` is a lower bound of any permutation of `x^a`. I.e. the list with the head element `y` put in front the `x` inserted ordered into `a` is sorted as well.

```
into (x:L, l:[L]): [L]
    -> inspect l
       case []  then [x]
       case y^a then
           if x <= y then x^y^a else y ^ x.into(a) end
       end
```

We claim that the function `into` returns a list which is a permutation of `x^a` and sorted. Before trying to prove these facts by induction let us look at the different cases separately.

The insertion of an element into the empty list maintains the permutation property.

```
all(x:L)
    proof
        x.into([]) = [x]
    ensure
        permutation ([x], x.into([]))
    end
```

The induction step requires to distinguish two cases because the function `into` contains an if expression for insertion into a nonempty list. We prove the permutation property separately for the two cases.

```
all(x,y:L, a:[L])
    require
        permutation(x^a, x.into(a))
        x <= y
    proof
         x.into(y^a) = x^y^a
    ensure
        permutation(x^y^a, x.into(y^a))
    end
```

```
all(x,y:L, a:[L])
    require
        permutation(x^a, x.into(a))
        not (x <= y)
    proof
        permutation(x^y^a, y^x^a)        -- module list
```

```
        permutation(y^x^a, y^x.into(a))  -- ind hypo/list
        proof
            x.into(y^a) = y^x.into(a)
            y^x.into(a) in {l: permutation(l,x.into(y^a))}
        ensure
            permutation(y^x.into(a), x.into(y^a))
        end
    ensure
        permutation(x^y^a, x.into(y^a))
    end
```

Having these two lemmas the proof by induction of the permutation property is easy.

```
all(x:L, a:[L])
    proof
        [] in {a: permutation (x^a, x.into(a))}

        all(y:L, a:[L])
            require
                permutation(x^a, x.into(a))
            proof
                x <= y or not (x <= y)

                x <= y       ==> permutation(x^y^a, x.into(y^a))
                not (x <= y) ==> permutation(x^y^a, x.into(y^a))
            ensure
                permutation(x^y^a, x.into(y^a))
            end

        a in {a: permutation (x^a, x.into(a))}
    ensure
        permutation (x^a, x.into(a))
    end
```

In order to prove the sortedness of the resulting list we have to make the same case split in the induction step.

```
all(x:L, a:[L])
    proof
        []  in {a: a.is_sorted ==> x.into(a).is_sorted}

        all(x,y:L, a:[L])
            require
                a.is_sorted ==> x.into(a).is_sorted
                (y^a).is_sorted
            proof
                x <= y or not (x <= y)

                require x <= y
                ensure  x.into(y^a).is_sorted end

                require not (x <= y)
                proof   y.is_lower_bound(x^a)
                        permutation(x^a, x.into(a))
                        (y^x.into(a)).is_sorted
                ensure  x.into(y^a).is_sorted end
            ensure
```

```
                x.into(y^a).is_sorted
            end

        a in {a: a.is_sorted ==> x.into(a).is_sorted}
    ensure
        a.is_sorted ==> x.into(a).is_sorted
    end
```

The insertion of one element into a sorted list has been the hard part. In order to sort the list completely we just have to step recursively through the elements and insert one by one into an initially empty list.

```
sorted (a:[L]): [L]
    -> inspect a
       case []  then []
       case h^t then h.into(t.sorted)
       end
```

The proofs of the permutation property and the sortedness is just an application of the standard template for doing induction proofs with some help at the induction step.

```
all(a:[L])
    proof
        [] in {a: permutation(a, a.sorted)}
        all(x:L, a:[L])
            require
                permutation(a, a.sorted)
            proof
                permutation(x^a, x^a.sorted)
                permutation(x^a.sorted, x.into(a.sorted))
                proof  x.into(a.sorted) = (x^a).sorted
                ensure permutation(x.into(a.sorted),
                                   (x^a).sorted) end
            ensure
                permutation(x^a, (x^a).sorted)
            end
        a in {a: permutation(a, a.sorted)}
    ensure
        permutation(a, a.sorted)
    end
```

```
all(a:[L])
    proof
        [] in {a: a.sorted.is_sorted}

        all(x:L,a:[L])
            require
                a.sorted.is_sorted
            proof
                x.into(a.sorted).is_sorted
            ensure
                (x^a).sorted.is_sorted
            end

        a in {a: a.sorted.is_sorted}
    ensure
```

```
        a.sorted.is_sorted
    end
```

Chapter 4

Proof Engine

4.1 Rules of the Proof Engine

The proof engine is based on a few very simple rules and knows only of universal quantification, existential quantification and implication.

4.1.1 Modus Ponens and Deduction Law

The modus ponens law says: Whenever the assertions

```
a

a ==> b
```

are valid with arbitrary boolean expressions `a` and `b` then we can conclude the validity of

```
b
```

The modus ponens law allows the proof engine to draw conclusions by following implications.

The deduction law allows us to prove implications. It says: In order to prove the implication

```
a ==> b
```

with arbitrary boolean expression `a` and `b` we can assume `a` and prove `b` under this assumption. I.e. the two assertions

```
a ==> b

require
    a
proof
    ...
ensure
    b
end
```

are equivalent i.e. if we can prove the latter we have proved the former.

The deduction law can be generalized to arbitrary implication chains. I.e. the proof engine is allowed to prove the chain

```
a ==> b ==> ... ==> z
```

by proving the assertion

```
require
    a; b; ...
proof
    ...
ensure
    z
end
```

4.1.2 Generalization of Variables

Assume that we want to prove the assertion

```
all(x,y,...) exp
```

I.e. we have to prove that `exp` is valid for all possible values of the variables.

Such a statement can be proved by assuming arbitrary values for the variables and proving the expression.

The proof engine proves a universally quantified statement by shifting the variables into the context and then proving the expression. If this is successful then the validity of the universally quantified assertions is concluded.

Furthermore the proof engine knows that in it can reorder the variables and bubble up variables i.e. it considers the two following assertions equivalent (with proper renamings of variables to avoid name clashes).

```
all(x,y) a ==> b ==> all(z) c

all(y,x,z) a ==> b ==> c
```

The last rule is used by the proof engine to transform universally quantified expressions to prenex normal form.

4.1.3 Specialization of Variables

If there is a proved statement of the form

```
all(x:X, y:Y,...) exp
```

the proof engine is allowed to generated valid asssertions by substituting the variables `x,y,...` by arbitrary expressions of consistent type.

The specialization can be partial if the expression `exp` is an implication where the first assumption contains only a part of the variables. E.g. if we have the proved assertion

```
all(x,y) f(x) ==> g(x,y)
```

and the expression `a` has a type compatible to the type of `x` then the proof engine can do a partial specialization and conclude the assertion

```
f(a) ==> all(y) g(a,y)
```

If the first premise in the implication chain does not contain any of the variables then partial specialization can be done as well. E.g. the assertion

```
all(a:BOOLEAN) false ==> a       -- ex falso quodlibet
```

can be partially specialized to

```
false ==> all(a:BOOLEAN) a
```

Example:

```
all(x,y,z) x or y ==> (x ==> z) ==> (y ==> z) ==> z
a in p or a in q

-- allows the partial specialization
(a in p or a in q) ==>
all(z) (a in p ==> z) ==> (a in q ==> z) ==> z
```

4.1.4 Existential Quantification

The existentially quantified assertion

```
some(x,y,...) exp
```

can be proved by substituting the variables `x,y,...` in the expression `exp` by some other expressions and proving the substituted expression. This method is called *finding a witness for the existentially quantified variables*.

If the above existentially quantified assertion has already been proved the proof engine is allowed to conclude the following assertion from it.

```
all(a:BOOLEAN) (all(x,y,...) exp ==> a) ==> a
```

The validity of this assertion should be obvious. If there exist some values for the variables `x,y,...` so that the expression `exp` is valid and for all values of the variables the validity of `exp` implies the validity of `a` then the assertion `a` is valid.

4.1.5 Function Expansion

The proof engine is allowed to expand function definitions. E.g. boolean negation is defined as

```
(not) (a:BOOLEAN) ->  a ==> false
```

Therefore the proof engine can expand the expression

```
not (a <= b)
```

to

```
a <= b  ==> false
```

4.1.6 Beta Reduction

Albatross has two kind of lambda expressions: function and predicate literals of the form

```
(x,y,...) -> exp

{x,y,...: exp}
```

The following expressions can be substituted by `exp2` where `exp2` is `exp` with all variables `x,y,...` substituted by the expressions `a,b,....`

```
((x,y,...) -> exp) (a,b,...)

{x,y,...: exp} (a,b,...)

(a,b,...) in {x,y,...: exp}
```

Note that `a in p` and `p(a)` are equivalent for predicates `p`.

4.2 How the Proof Engine Works

A context of the proof engine is a sequence of variables and a sequence of assumptions and a sequence of proved assertions. Contexts can be stacked. A new inner context is opened by shifting variables and assumptions into the context. Within the context assumptions are treated as proved statements. By leaving a context the variables and assumptions have to be discharged.

4.2.1 Proof of an Assertion Feature

An assertion feature has the general form:

```
all(a,b,...)
    require
        assumption_1
        assumption_2
        ...
    proof
        intermediate_1
        intermediate_2
        ...
    ensure
        conclusion_1
        conclusion_2
    end
```

where all assumptions and conclusions are expressions and all intermediate assertions are expressions or nested assertion features with or without new variables.

The proof engine proves an assertion feature by the following algorithm.

1. Open an new context
2. Shift all variables into the context.

3. For all assumptions: Prove the preconditions of the assumption and shift the assumtion into the context.

4. For all intermediate assertions:

   (a) If it is an expression then prove all preconditions and prove the expression and add it to the context.

   (b) If it is an assertion feature then apply this algorithm recursivly and add the result to the context.

5. For all conclusions prove the preconditions of the conclusion and the conclusion and add the conclusion to the context.

6. Leave the context and add all conclusions with properly discharged variables and assumtions to the outer context

4.2.2 Prove an Assertion

Proving of any assertion (precondition or assertion expression) is done by the following algorithm:

1. Forward close the context by forward reasoning.

2. Enter the assertion as deep as possible i.e. split universal quantification and add the corresponding variables and split implications and add assumptions.

3. Directly prove the goal. In case of success the proof is done. In case of failure continue.

4. Prove the goal by backward reasoning.

5. Reverse the entering process and discharge the proved goal.

4.2.3 Directly Prove a Goal

The proof engine tries to directly prove a goal after it has been fully entered i.e. the goal is neither a universally quantified expression nor an implication.

A goal is directly provable if one of the following holds:

1. The goal is already in the context.

2. A schematic assertion can be found in the context together with a substitution so that the substitution applied to the schematic assertion makes it identical with the goal.

3. The goal is an existentially quantified assertion and a witness expression can be found in the context to prove the existential quantification.

4. The goal is an equality of the form

```
f(a1,a2,...) = f(b1,b2,...)
```

and the assertions

```
a1 = b1
a2 = b2
...
```

are already in the context.

4.2.4 Entering and Discharging

An assertion expression has the general form

```
all(x,y,...) a ==> b ==> ... ==> z
```

where the universal quantification and the premises of the implication chain are optional. If no universal quantification and no premises are present then the assertion expression is the unsplittable expression `z` and the enter and the discharge procedures have nothing to do.

The entering procedure consists in

1. Shift the variables `x,y,...` into the context.
2. Shift the assumptions `a,b,...` one by one into the context.
3. Forward close the context by forward reasoning.
4. If the target `z` can be entered apply the procedure recursively

After entering the assertion expression there is a goal `z` to be proved which cannot be entered any more. The goal (or a witness for it) can either be found in the context or can be proved by backward reasoning (or the proof fails).

Lets assume that the prove of the target goal `z` has succeeded. Then the entering process has to be reversed by the discharge procedure. Since the entering procedure is recursive we have to consider the general case that the original assertion expression has the form

```
all(x,y,...) a ==> b ==> ... ==> all(t,u,...) z0
```

The proof engine normalizes the original assertion to

```
all(x,y,...,t,u,...) a ==> b ==> ... ==> z0
```

with proper renaming to avoid name clashes. Furthermore all unused variables are eliminated and the remaining variables are sorted to reflect the order in which they appear in the implication chain.

4.2.5 Forward Reasoning

Forward reasoning is the process of drawing conclusions from the assertions within a context. A context is forward closed by applying the following rules as long as possible:

1. Find a pair of assertions of the form `a` and `a ==> b` and apply the moduls ponens law by adding `b` to the context.
2. Find a pair of assertions of the form `a` and `all(x,y,...) p ==> ...` where `p` is not a pure variable and a (partial) substitution of the variables so that `p` becomes identical to `a` if the substitution is applied. Add the (partially) specialized assertion to the context.
3. Find an existentially quantified assertion `some(x,y,...) exp` and add the assertion

   ```
   all(a) (all(x,y,...) exp) ==> a
   ```

   to the context.
4. Find an assertion where an evaluation is possible. Add the evaluated assertion to the context.

Some restrictions apply to the application of the first and the second rule.

1. The (partial) specialization of a schematic assertion of the form

   ```
   all(x,y,...) a ==> b ==> ... ==> z
   ```

   is called intermediate and remains intermediate until the final conclusion `z` has been added fully specialized to the context.
2. An intermediate assertion is never used as the premise in the modus ponens rule.
3. The final fully specialized conclusion of an intermediate assertion is added to the context only if the conclusion is simpler than all premises. A term `a` is simpler than a term `b` if its expression tree has less nodes than the expression tree of the term `b` after all function expansions have been done.

These restrictions guarantee the termination of forward reasoning.

4.2.6 Backward Reasoning

If an assertion has been fully entered and is not directly provable then the proof engine starts to prove the goal by backward reasoning.

1. Find and generate backward rules. A valid backward rule is an implication chain of the form

   ```
   p1 ==> p2 ==> ... ==> goal
   ```

which has not yet been used in the proof.

2. Find one backward rule for which all premises can be proved. In case of success the proof is done by applying the modus ponens rule, in case of failure the proof fails.

There are two ways to generate backward rules:

1. Find a schematic assertion of the form

   ```
   all(x,y,...) p1 ==> p2 ==> ... ==> z
   ```

   where `z` is not a variable and can be unified with the goal by some substitution of the variables. The fully specialized assertion is a valid backward rule. Restriction: If some of the premises (after the substitution and with all functions expanded) are more complicated than the goal (with all functions expanded) and the schematic rule has already been used before in the proof, then the generated backward rule is not valid.

2. If `eval` is a valid evaluation of the goal then

   ```
   eval ==> goal
   ```

   is a valid backward rule.

4.2.7 Evaluation

Universally quantified expressions, implications and conjunctions i.e. expressions of the following forms are never evaluated.

```
all(x,y,...) exp

a and b

a ==> b
```

Ways to evaluate an expression:

1. Expansion of the toplevel function. If the toplevel function is a function term then do a toplevel expansion of the function term.

2. If the toplevel function does not have a definition and the expression is neither universally quantified nor an implication nor a conjunction then the expression with all functions expanded recursively is a valid evaluation.

3. Beta reduction of the toplevel term.

4. If the expression has the subterms `a1,a2,...` i.e. it has the form `f(a1,a2,...)` and there are the assertions

   ```
   a1 = b2
   a2 = b2
   ...
   ```

in the context where the right hand side of the equation is simpler than the left hand side (after full function expansion), then `f(b1,b2,...)` is a valid evaluation of the expression.

Chapter 5

Language Reference

5.1 Structure of Albatross Programs

Modules are compilation units. Each module has an implementation file with the extension `.al` and an interface file with the extension `.ali`. The user of a module has access only to the elements declared in the interface file.

A collection of modules in the same directory form an Albatross package or an Albatross library. The compiler works on Albatross packages. Within a directory of Albatross source files the compiler offers you the following basic commands:

```
alba init       -- initializes the directory
alba status     -- displays modules which need recompilation
alba compile    -- compiles all modules which require recompilation
alba help       -- get help about commands, options and arguments
```

I.e. the albatross compiler automatically keeps track of dependencies between modules.

Each module has to start with a declaration which states the used modules.

```
-- some source file
use
    module_1
    module_2
    ...
end

...
```

The order of the used modules is irrelevant. The modules used by `module_1` and `module_2` are used implicitly.

The name of a module is the filename of the module's source file without its extension. I.e. the module name of the module having the source files `boolean.al` and `boolean.ali` is `boolean`.

Modules of the same package are used by name. Modules of other packages are used by their fully qualified names. E.g. a module with the usage declaration

```
use
    alba.base.boolean
    alba.base.predicate
    graphic
end
```

uses the modules `boolean` and `predicate` of the package `alba.base` and the module `graphic` of the same package.

The package name is the name of the directory where it resides. The package `alba.base` has to reside in a directory named `alba.base`.

The Albatross compiler searches for packages within its search path. Search paths can be given to the compiler on the command line

```
alba -I path/to/package_1 -I path/to/package_2 ... compile
```

or by defining an environment variable called `ALBA_LIB_PATH`. If the environment variable has the content

```
path/to/lib_1:path/to/lib_2:path/to/lib_3:...
```

then the Albatross compiler searches for libraries in all these paths.

The paths on the command line override the paths of the environment variable.

The locations of commonly used libraries should be given by the environment variable and the location of libraries/packages used only for a specific program should be given on the command line.

5.2 Modules

A module is the basic compilation unit. Each module has an implementation file with the name `name.al` an an optional interface file with the name `name.ali` where `name` is the name of the module.

The interface file just defines the public view of a module. Therefore it cannot introduce anything which does not appear in the corresponding implementation file.

A module without an interface file cannot be used by other modules.

A module file consists of a usage header and a sequence of declarations.
Syntax:

```
module:       [use_block] declarations

use_block:    use module_list end

module_list:  {one_module separator ...}+

one_module:   {identifier '.' ...}+

declarations: {declaration [separator] ...}*

declaration:  class_declaration
|             formal_generic
|             assertion_feature
```

```
|             named_feature

formal_generic:     uidentifier ':' type
```

Rules and validity:

1. No circularity is allowed in the usage of modules.
2. An interface file can use only modules which have been used in the implementation file. Usage can be implicit. E.g. if module `a` uses module `b` and module `b` uses module `c` then module `a` uses module `c` implicitly even if module `c` has not been mentioned in the usage block of module `a`.
3. A used module of the same package must not be prefixed with the package name.

A separator can be a semicolon or a newline (for details on newlines as separators see the chapter lexical conventions 5.10).

A used module in the usage block has the form `a.b.c.m` where `m` is the name of the module and `a.b.c` is the name of the package where the module resides.

A formal generic is a type variable. The declaration `A:TP` declares the type variable `A` with the concept type `TP`. The type variable `A` can be replaced by any type which satisfies the concept (i.e. inherits from) `TP`.

The name scope of a formal generic is its module.

Examples:

The module `boolean` of the basic library e.g. has the following interface file:

```
-- file: boolean.ali

immutable class BOOLEAN end

false: BOOLEAN
true:  BOOLEAN
(==>) (a,b:BOOLEAN): BOOLEAN
(and) (a,b:BOOLEAN): BOOLEAN
(or)  (a,b:BOOLEAN): BOOLEAN

(not) (a:BOOLEAN): BOOLEAN)   -> a ==> false
(=)   (a,b:BOOLEAN): BOOLEAN) -> (a ==> b) and (b ==> a)

all(a,b:BOOLEAN)
    ensure
        (not a ==> false) ==> a
        a and b ==> a
        a and b ==> b
        ...
    end
```

The module declares the type `BOOLEAN` as immutable. I.e. all objects of type `BOOLEAN` cannot be modified.

If a class has the same name as the module, just in uppercase, then the class is available to the users without any qualification. If a class has a name different from the module name it can be accessed by the users of the module only by

qualifying the class name with the modulename. E.g. the class `A` declared in the module `x` has to be used outside the defining module as `x.A`. Sometimes it might even be necessary to add the package name (e.g. `p.x.A`) to disambiguate the situation.

Furthermore it defines the constants `true` and `false`. No definition of these constants are visible to the user. The implementation file might have a definition of these constants, but they are hidden from the user.

The three binary boolean functions implication `==>`, conjunction `and` and disjunction `or` are declared without definition which means their definition is not available to the user.

The functions negation `not` and boolean equivalence `=` are declared with a definition. Since the definition is visible the compiler can expand any expression of the form `not (x + y = z)` to `(x + y = z) ==> false`

The assertion declaration states valid assertions about booleans. In the interface file the assertions are just stated. The proofs can be found in the corresponding implementation file `boolean.al`.

The first assertion `(not a ==> false) ==> a` gives us the opportunity to prove any assertion `a` by assuming `not a` and deriving from it `false`.

The module `any` of the basic library has the following interface file:

```
-- file: any.ali
use boolean end

deferred class ANY end

G: ANY

(=)  (a,b:G): BOOLEAN    deferred end
(/=) (a,b:G): BOOLEAN -> not (a = b)

all(a:G) deferred ensure a = a end
all(a:G) ensure a /= a ==> false end

immutable class boolean.BOOLEAN
inherit         ANY end
```

The module declares the class `ANY` as a deferred class which means that it can have deferred functions and/or assertions which have to be defined in other classes which inherit from the class `ANY`.

The module defines a formal generic `G`. Formal generics are type variables. Any type can be substituted for `G` as long as the type inherits from the class `ANY`.

The module `any` has the purpose to define equality. Therefore it declares a deferred function named `=` which compares two objects of the same type. Any descendant of `ANY` has to define the function or redeclare it as deferred. Furthermore it declares an assertion stating reflexivity of equality. Since the assertion is deferred any descendant of the class `ANY` has to either prove this statement or restate it as deferred.

Inequality `/=` is defined in the module `any` as an effective function. This function is inherited to any descendant.

Furthermore the module `any` states that it has proved the fact that `a /= a ==> false` is valid for any `a`. Any descendant of `ANY` can take this fact for granted.

Note that the class name `BOOLEAN` is qualified with the module name in the inheritance class despite the fact that the class `BOOLEAN` has the same name as the corresponding module. The qualification is necessary in this case because otherwise the compiler would generate a new class `BOOLEAN` of the module `any` which is different from the corresponding class of the module `boolean`.

The module `any` states with this inheritance clause that the class `BOOLEAN` can inherit the class `ANY` i.e. that it has a function `=` which is reflexive.

The module `boolean` cannot state this inheritance relation because it cannot use the module `any` since the module `any` already uses the module `boolean` and no circularity is allowed.

5.3 Classes

Syntax:

```
class_declaration:  [header_mark] class
                    class_name [class_generics]
                    [creator_clause]
                    [inherit_clause]
                    end

header_mark:        immutable | deferred | case

class_name:         {identifier '.' ...}* uidentifier

class_generics:     '[' {uidentifier ',' ...}+ ']'
```

Rules and validity:

1. In a class declaration only formal generics can be used which have already been declared in the surrounding module.
2. A redeclaration of a class must have the same header mark, the same number of generics and the formal generics must have the same concepts as the previous declaration.
3. All deferred functions and assertions of a class have to be declared in the same module as the first declaration of the class.
4. If a class has already been used as a parent in an inheritance clause then no more deferred functions and assertions can be declared for the class.

The first declaration of a class in a module does not have an inheritance clause i.e. it has one of the forms

```
immutable class BOOLEAN end         -- file: boolean.al[i]

deferred class ANY end              -- file: any.al[i]
```

```
deferred class PARTIAL_ORDER end   -- file: partial_order.al[i]

immutable class FUNCTION[A,B] end  -- file: function.al[i]
```

After the class has been declared with its potential formal generics, features and assertions can be introduced to the class.

Classes may add inheritance clauses if they also declare the appropriate features and assertions.

A class declaration where the classname is unqualified always declares a class which belongs to the current module. The first declaration of a class within a module introduces a new class.

If a class has the same name as the module (just in upper case) then the class can be used by other modules to declare types without any qualification (however qualification can be used to remove ambiguities).

A class having a name different from the module name can be used outside the defining module to define types only by adding the module name to the class name in front of the class name.

A class can redeclared as often as needed (usually to add inheritance clauses) as long as the redeclaration is consistent with the original declaration.

Classes can be redeclared in modules different from the original module, but in the reclaration the class name has to be prefixed with the name of the original module. Without the module name a new class having the same name is declared. The following example illustrates this.

```
-- file a.ali
immutable class A end   -- declares a new class 'A' in module 'a'

-- file b.ali
immutable class A end   -- declares a new class 'A' in module 'b'

immutable class a.A end -- redeclares the class 'A' of the module 'a'
```

5.4 Creators

Syntax:

```
creator_clause:    create  { creator [separator] ...}+

creator:           feature_name ['(' formals ')' ]
```

Definition: A base creator is a creator which has no argument depending on the surrounding case class. The other creators are recursive.
Rules and validity:

1. There has to be at least one base creator.
2. All base creators have to occur before the recursive creators.
3. The creators can use only formal generics of the surrounding class.
4. All formal generics have to appear in at least one of the constructors in a nonrecursive argument.

5.5 Inheritance

Syntax:

```
inherit_clause:    inherit {parent [separator] ...}+

parent:            [ghost] type
```

Rules and validity:

1. Immutable types cannot be used as parents.
2. A parent can involve only formal generics of the inheriting class.
3. The inheritance relation must be acyclic.
4. All deferred functions and assertions of the parent have to be present in the inheriting class, either redeclared as deferred or redeclared with a definition or proof respectively.
5. All effective features of the parent which have already a declaration in the inheriting class must be consistent with the definition in the parent.
6. If a deferred function of the parent is present in the inheriting class as a ghost function then the parent must be a ghost type.

5.6 Types

Syntax:

```
type:   simple_type
|       predicate_type
|       function_type
|       list_type
|       '(' {type ',' ...}+ ')'

simple_type:       class_name [actual_generics]

actual_generics:   '[' {type ',' ...}+ ']'

predicate_type:    type '?'

function_type:     type '->' type

list_type:         '[' {type ',' ...}+ ']'

-- '?' has higher precedence than '->'
```

Rules and validity:

1. The number of actual generics has to coincide with the number of formal generics of the class.
2. Each actual generic has to satisfy the concept of the corresponding formal generic of the class.

Examples:

```
(A,B,C)       -- tuple with three types
              -- parsed as (A,(B,C))
              -- shorthand for TUPLE[A,TUPLE[B,C]]

A -> B        -- shorthand for FUNCTION[A,B]

(A,B) -> C    -- shorthand for FUNCTION[(A,B),C]

A -> B -> C   -- shorthand for FUNCTION[A, FUNCTION[B,C]]
              -- '->' is right associative

A?            -- shorthand for PREDICATE[A] i.e. a set of
              -- elements of type A

(A,B)?        -- shorthand for PREDICATE[(A,B)] i.e. a set of
              -- tuples of type (A,B)

A -> B?       -- shorthand for FUNCTION[A,PREDICATE[B]]

(A -> B)?     -- shorthand for PREDICATE[FUNCTION[A,B]]

[A]           -- shorthand for LIST[A]

[A,B->C]      -- shorthand for LIST[TUPLE[A,FUNCTION[B,C]]]
```

The basic library has the modules `tuple`, `predicate` and `function`. If any of the corresponding types is used the module has to appear directly or indirectly in the use block of the using module.

5.7 Named features

Syntax:

```
named_feature: feature_name ['(' formals ')' ]
               [':' [ghost] type]
               [feature_body]

feature_name:  identifier
|              '(' operator ')'
|              number
|              true
|              false

formals: {formal_argument ',' ...}+

formal_argument:  identifier [':' type]

feature_body:  '->' expression
|              '='  expression
|              [require_block]
               [feature_implementation]
               [ensure_block]
               end
```

```
require_block: require {expression separator ...}+

ensure_block:  ensure  {expression separator ...}+

feature_implementation:  deferred
```

Rules and validity:

1. In a named feature at least one of the optional components must be present.
2. In the feature body at least one of the optional components must be present. I.e. a feature body cannot consist of the keyword `end` only.
3. If the feature body is `-> expression` then formal arguments must be present and the expression must not have any preconditions i.e. the function is total.
4. If the feature body is `= expression` then the feature is a constant and no formal arguments are allowed and the expression must not have any preconditions.
5. If the feature name is an operator there must be one formal argument for unary operators and two formal arguments for binary operators. Note that most of the operators can be unary and binary.
6. If the implementation of a function is not computable (e.g. if it involves universal or existential quantification) then the keyword `ghost` must be present in the result type.
7. The postcondition of a function might be of the form `Result = exp`. If the expression contains non computable elements then the function must be declared as ghost function.
8. If the postcondition specifies only properties of the return value then the function has to be declared as ghost function and the proof engine must be able to prove the existence and uniqueness of the return value.

Examples:
A simple declaration without body:

```
my_function (a,b:A, d:D): RT
```

This is a valid declaration of a function having the signature `(A,A,B):RT`. The function has no body. Within an implementation file such a feature is practically useless because you cannot do anything with it. In an interface file this declaration can be useful if assertions are given which supply useful properties of the function.
A constant and a binary operator without body and a unary and binary operator with body:

```
false: BOOLEAN

(==>) (a,b:BOOLEAN): BOOLEAN

(not) (a:BOOLEAN): BOOLEAN   -> a ==> false

(=)   (a,b:BOOLEAN): BOOLEAN -> (a ==> b) and (b ==> a)
```

The empty set and the universal set defined as constants with body:

```
G: ANY

0: G? = {x: false}
1: G? = {x: true}
```

A ghost function involving non computable elements in its definition:

```
PO: PARTIAL_ORDER

is_lower_bound (a:PO, p:PO?): ghost BOOLEAN
    -> all(x) x in p ==> a <= x
```

Declaration of a function with a precondition:

```
A: ANY; B: ANY

value_at (f:A->B, a:A):B -> f(a)  -- ILLEGAL!!

value_at (f:A->B, a:A):B
    require
        a in f.domain
    ensure
        Result = f(a)
    end
```

All functions in Albatross are potentially partial. Therefore the first declaration is not valid because the expression has a precondition. The second declaration is valid because it makes the precondition explicit.
Declaration of a function by using properties:

```
A:ANY; B:ANY

inverse (f:A->B): ghost (B->A)
    require
        f.is_injective
    ensure
        Result.domain = f.range
        all(x) x in f.domain ==> Result(f(x)) = x
    end
```

The inverse of a function has to be declared as a ghost function because the return value of `inverse` is specified in the postcondition with properties and not directly with a computable formula.

Note: If the return value of a function is given by properties then proof engine must be able to prove that the return value exists and that it is unique.

5.8 Assertions

Syntax:

```
assertion_feature: all '(' formals ')'
                   [require_block]
                   [proof_block]
                   ensure_block
                   end

proof_block:       proof {proof_expression separator ...}+

proof_expression:  expression
|                  assertion_feature
|                  [require_block]
                   proof_block
                   ensure_block
                   end
```

Rules and validity:

1. All assertions must be provable by the proof engine.

Examples:

```
all(p:G?)
    require
        some(x) x in p
    proof
        all(x) x in p ==> p /= 0
    ensure
        p /= 0
    end
```

```
all(p:G?)
    require
        p /= 0
    proof
        not (some(x) x in p) ==> false
    ensure
        some(x) x in p
    end
```

```
all(p,q:G?)
    require
         p /= 0
    proof
         some(x) x in p
         all(x)
             require
                 p(x)
             proof
                 x in p + q
                 some(x) x in {x: x in p + q}
             ensure
                 p + q /= 0
             end
    ensure
```

```
        p + q /= 0
    end
```

5.9 Expressions

Syntax:

```
expression: simple_expression
|           operator_expression
|           f_call
|           oo_call
|           tuple
|           function_literal
|           predicate_literal
|           list
|           listed_set
|           quantified
|           typed
|           '(' expression ')'

simple_expression:
            identifier | number | false | true | Result

operator_expression:
            expression binary_operator expression
|           unary_operator expresssion

f_call:     expression '(' actuals ')'

oo_call:    expression '.' identifier ['(' actuals ')']

tuple:      '(' expression ',' {expression ',' ...}+ ')'

actuals:    {expression ',' ...}+

function_literal:
            '(' formals ')' [':' type] '->' expression
|           agent '(' formals ')' [':' type]
                [require_block]
                ensure_block
                end

predicate_literal:
            '{' formals ':' expression '}'

listed_set: '{' {expression ',' ...}+ '}'

list:       '[' {expression ',' ...}* ']'

quantified: quantifier '(' formals ')' expression

quantifier: all | some

typed:      expression ':' type

binary_operator:
```

```
            '+' | '-' | '*' | '/' | and | or | '==>' |
            '^' |
            '<=' | '<' | '>' | '>=' |
            '=' | '/=' | '~' | '/~' |
            free_operator

unary_operator:
            not | old | '+' | '-' | '*' | free_operator
```

Precedence and associativity:

Ambiguities are resolved according to the following precedences and associativities starting with the highest precedence.

```
'.'                         -- left
not, old
free_operator               -- left
'^'                         -- right
'*', '/'                    -- left
'+', '-'                    -- left
'<=', '<', '>=', '>', '=', '/=', '~', '/~', in, /in
and, or
'==>'                       -- right
```

Examples:

Functions can be called in functional or in object oriented notation. The following expressions are equivalent

```
-- functional                    object oriented

f(x,y,...)                       x.f(y,...)

f(x)                             x.f

f(f(f(f(x))))                    x.f.f.f.f
```

If the argument of a call is a tuple then nested parentheses are not necessary, i.e. the following simplified notation is possible (only for functional notation).

```
-- detailed                      simplified
r((x,y))                         r(x,y)
```

Tuples are parsed with right association.

```
-- tuple                         parsed as
(a,b,c,...)                      (a,(b,(c,...)))
```

The quantifiers and the arrow -> are greedy i.e. they span to the right as far as possible. To limit the span parentheses can be used.

Types are greedy as well, i.e. they span to the left as far as possible.

A listed sets and lists are just a shorthand notations. The parser does the following expansion

```
-- shorthand                     expansion
{a,b,...}                        a.singleton + b.singleton + ...

[a,b,...,z]                      a ^ b ^ ... ^ z ^ []
```

5.10 Lexical Conventions

There are two types of comments. The first one starts with the sequence `--` followed by a blank and spans to the end of the line. The second one begins with the sequence `{:` and ends with `:}` and spans an abitrary amount of text. The second comment type can be nested. Comments are completely ignored by the compiler.

A separator is either the semicolon or a newline between an endtoken and a begintoken.

```
begintoken:     identifier
                uidentifier
                number
                require
                ensure
                proof
                true
                false
                not
                '('
                '['
                '{'

endtoken:       identifier
                uidentifier
                number
                true
                false
                ')'
                ']'
                '}'
```

Note that no operator can be an endtoken because Albatross does not have postfix operators. Furthermore nearly all operators cannot be starttokens either. Therefore if a new expression starts with a unary operator the expressions has to be either parenthesized or the previous expression has to be terminated with a semicolon.

```
identifier:     [a-z][a-z0-9_]*

uidentifier:    [A-Z][A-Z0-9_]*

number:         [0-9]+

free_operator: [+-/*<>=~|^]+
```

The following identifiers are reserved words in Albatross:

```
all             and             as              assert
case            check           class           create
deferred        do
else            elseif          end             ensure
false           feature         from
ghost
if              immutable       import          in              inherit
inspect         invariant
```

```
local
not           note
old           or
proof
redefine      rename       require
some
then          true
undefine      use
variant
while
```