4 rue Léonard de Vinci
BP 6759
F-45067 Orléans Cedex 2
FRANCE
http://www.univ-orleans.fr/lifo

Rapport de Recherche

A Formal Programming Model of Orléans Skeleton Library

Noman Javed
Frédéric Loulergue

Rapport n^o **RR-2011-06**

A Formal Programming Model of Orléans Skeleton Library

Noman Javed LIFO, University of Orléans, France
Noman.Javed@univ-orleans.fr

Frédéric Loulergue
LIFO, Université d'Orléans, France
Frederic.Loulergue@univ-orleans.fr

April 2011

Abstract

Orléans Skeleton Library (OSL) is a library of parallel algorithmic skeletons in C++ on top of MPI. It provides a structured approach towards parallel programming. Skeletons in OSL are based on the bulk synchronous parallelism model. In this paper we present formal semantics of OSL: a programming model and its properties proved with the Coq assistant.

Contents

1 Introduction

If parallel architectures are now widespread, it is not yet the case for parallel programming. For distributed memory or shared memory machines, quite low level techniques such as PThreads or MPI are still widely used. To ease the programming of parallel machines, more structured approaches are needed. Algorithmic skeletons [5, 14, 15] that can be seen as higher-order functions implemented in parallel, offer a programming style in which the user combines patterns of parallel algorithms to build her application. Bulk synchronous parallelism [17] is another structure approach of parallelism that provides a simple and portable performance model.

Our Orléans Skeleton Library [11] is a library of data parallel algorithmic skeletons that follows the BSP model. OSL is a library for the C++ language and it uses expression template techniques as an optimisation mechanism.

In order to make this kind of library more reliable, and to be able to prove the correctness of programs written in OSL, we plan to provide formal semantics for the programming model of OSL (ie the semantics that is presented to the user of the library) as well as for the execution model of OSL (ie the semantics of the implementation of the library), and to prove the equivalence of these semantics using the Coq proof assistant[16, 2]: this will give a good confidence in the implementation of OSL.

In this paper we first present informally the OSL library (section 2) before compare this work with related papers (section 3). Then we describe the formal programming model of OSL using the Coq proof assistant (section 4) before we conclude (section 5). We also give a short introduction to the Coq proof assistant in appendix (section A).

2 An Overview of Orléans Skeleton Library

Orléans Skeleton Library is a library of data parallel algorithmic skeletons on distributed vectors. It is implemented in C++ currently on top of MPI. C++ templates are heavily used for the implementation of the OSL for taking advantage of the functional programming paradigm. The goal of the skeleton approach is to provide a small set of basic parallel patterns. The applications are developed by finding the appropriate combination and composition of these skeletons.

In the BSP model, the number of memory-processor pairs is fixed during execution. This value is accessible to the programmer, it is named osl::**bsp_p**. These pairs are interconnected in such a way that point-to-point communications are possible. A global synchronisation unit is available in a BSP computer. The execution of a BSP program is a sequence of super-step, each one being composed of a phase where each processor computes using only the data it holds, a phase where processors exchange data and a synchronisation barrier that guarantees the completion of data exchange before the start of a new super-step. The other the other BSP parameters are respectively osl::**bsp_g** (network bandwidth), osl::**bsp_l** (synchronisation time), and osl::bsp_r which is a measure of the processors computing power. All parameters, but **bsp_p**, are obtained by a benchmark program called `oslprobe`.

The data structure used at the base of OSL is distributed array. The data is distributed among the processors at the time of the creation of the array. DArray is implemented as a template class. Thus a variable of type DArray<T> is a distributed array with

Type / Signature	Notation / Informal semantics
DArray<T> (sequential view)	$[t_0, \ldots, t_{t.size-1}]$
DArray<W> map(W f(T), DArray<T> t)	map$(f,\ [t_0, \ldots, t_{t.size-1}]) = [f(t_0), \ldots, f(t_{t.size-1})]$
DArray<W> zip(W f(T,U), DArray<T> t, DArray<U> u)	zip$(f,\ [t_0, \ldots, t_{t.size-1}],\ [u_0, \ldots, u_{t.size-1}]) =$ $[f(t_0, u_0), \ldots, f(t_{t.size-1}, u_{t.size-1})]$
<T> reduce(T⊕(T,T), DArray<T> t)	reduce$(\oplus,\ [t_0, \ldots, t_{t.size-1}]) = t_0 \oplus t_1 \oplus \ldots \oplus t_{t.size-1}$
DArray<Vector<T> > getPartition(DArray<T> t)	getPartition$([t_0, \ldots, t_{t.size-1}]) =$ $\langle\ [t_0^0, \ldots, t_{l_0-1}^0]\ , \ldots,\ [t_0^{p-1}, \ldots, t_{l_{p-1}-1}^{p-1}]\ \rangle$
DArray<T> flatten(DArray<Vector<T> > t)	flatten$(\langle\ [t_0^0, \ldots, t_{l_0-1}^0]\ , \ldots,\ [t_0^{p-1}, \ldots, t_{l_{p-1}-1}^{p-1}]\ \rangle) =$ $[t_0, \ldots, t_{t.size-1}]$
DArray<T> permute(int f(int), DArray<T> t)	permute$(f,\ [t_0^0, \ldots, t_{l_0}^0]) = [t_{f(0)}^0, \ldots, t_{f(l_0-1)}^0]$
DArray<T> shift(int dec, T f(T), DArray<T> t)	shift$(d,\ f,\ [t_0^0, \ldots, t_{l_0}^0]) =$ $[f(0), \ldots, f(d-1), t_0, \ldots, t_{t.size-1-d}]$

Figure 1: OSL Data Structure and Skeletons

elements of type T. As there are **bsp_p** processors in a BSP machine, a distributed array consists of **bsp_p** partitions evenly distributed (the partitions on the processors with low processor identifiers may have one more element than the processors with high processors identifiers).

Figure 1, gives the informal notations for distributed arrays and an informal semantics for the main OSL skeletons. In this figure, **bsp_p** is noted p.

A distributed array can be seen as a usual array. map (resp. zip) is the usual combinator to apply a function to each element of a distributed array (resp. of two distributed arrays). The first argument of both map and zip could be either a pointer function, or a functor either extending std::unary_function or std::binary_function.

getPartition exposes the partitioning of a distributed array, transforming a distributed array of type DArray<T> into a distributed array (containing one vector by processor) of type DArray<Vector<T> >. The inverse operation of getPartition is flatten.

reduce is a parallel reduction with a binary *associative* operator $\oplus$. Communications are needed to execute a reduce. permute and shift are communication functions. permute moves the content of the distributed array, hence redistributes the array, according to a permutation function f on the interval $[0, t.size - 1]$. shift is used to shift elements on the right (the case shown in the figure) or the left depending on the sign of its first argument. The missing values, at the beginning or the end of the array, are given by function f.

3 Related Work

There are many proposals for skeletal parallelism. A recent survey is [9]. Here we focus on work on formal semantics. Eden [10] and BSML [7, 3] are two parallel functional languages that are often used for implementing algorithmic skeletons. The latter has a mechanised formal semantics. Moreover a new algorithmic skeleton called BH as been implemented and its implementation proved correct using the Coq proof assistant. This BH skeleton is used in a framework for deriving programs written in BH from specification [8].

Some other work focus on algorithmic skeleton libraries, to our knowledge none is formalised and the properties of the semantics verified using a proof assistant.

[6] is a data-parallel calculus of multi-dimensional arrays, but no implementation was released. [1] is a formal semantics for the Lithium algorithmic skeleton language: it differs from OSL as it is a stream-based language. The work proposes in a single formalism a programming model and a (high-level) execution model.

The semantics of the Calcium library is described in [4] and further extended in a shared memory context to handle exceptions [13]. In [4], the focus in on a programming model semantics (operational semantics) as well as a static semantics (typing) anf the proof of the subject reduction property (the typing is preserved during evaluation). In this work the semantics of the skeletons are detailed, but not the semantics of what the authors call the "muscles" ie the sequential arguments of the skeletons (the semantics of the host language of the library, in the particular case Java). The set of skeletons of Calcium includes a set of task parallel skeletons, which contains, among others, skeletons that gives a sequential control but at the global level of all the parallel program: these skeletons are parallel because their branches or bodies are parallel (conditionals and while/for loops). In OSL we mix the skeletons with the usual constructs of the host C++ language to write the sequential control flow are the global level of the parallel program. The remain of the set of skeletons in Calcium are data-parallel skeletons including map, and divide-and-conquer skeletons. The map skeleton, for example, is however different from our map. The OSL map is more similar to map functions in functional programming as it takes two arguments: a function f to be applied to each element of the collection l which is the second argument. In functional programming this collection is a list, in OSL it is a distributed array. In Calcium the map skeleton takes two additional functions: one that describes how the input collection is cut into pieces and another function that describes how the pieces (obtained by applying f to the previous pieces) are combined together to form the output collection.

4 OSL Mechanised Semantics: Programming Model

We now present how we modelled the programming model of OSL using the Coq proof assistant. We first explain how the modelling of the data structure of distributed arrays and of the syntax is done. We then present a big-step semantics and its properties.

4.1 Distributed Arrays

First of all we need to model the parallel data structure of our OSL library: the distributed arrays. The content of a distributed array can be seen as a usual sequential array plus information about its distribution. In Coq we model the content of the arrays by lists. The distribution is modelled by a data structure similar to lists but with the size of the collection inside the type: vectors. A vector of type vector A n has size n and contains values of type A. A distribution is a vector of *naturals*, as it is the number of elements per processor that is the content of this distribution. The size of a vector of distribution is Bsp.p, the number of processor of the BSP machine. To cleanly formalise the fact that the syntax and semantics is parametrised by the number of processors of the parallel machine, the semantics is a functor, ie a module that takes as argument another module. This argument module has the following type:

```
Module Type BSP_PARAMETERS.
  Parameter p : nat.
End BSP_PARAMETERS.
```

This allows us to instantiate the functor with a module containing a specific value for Bsp.p in order to write examples and execute our semantics within Coq.

The type of distributed array is a record type:

```
Record distributedArray (A:Type) := mkDistributedArray {
  distributedArray_data : list A;
  distributedArray_distribution: vector nat Bsp.p;
  distributedArray_invariant:
    List.length distributedArray_data = globalSize distributedArray_distribution
}.
```

This type contains the two fields already described: the content of the parallel vector and the distribution of this content on the processor. However there is a third field: a proof that the value is indeed a coherent representation of a parallel vector: the sum of the elements of the distribution (computed using the function globalSize, omitted here) should actually be the length of the content list.

Values of this type are a kind of inner representation of distributed arrays that the user of the Orleans Skeleton Library could not used directly. She will be given a syntax for writing OSL programs.

4.2 Syntax and Typing

As in [4], we would like to model the semantics of our semantics without being obliged to model the whole syntax of the host language. The language of the Coq proof assistant can be seen as a pure functional programming language plus ways to express logical properties. Therefore to have the semantics of the host language as a black box we could model only the input/output behaviour of the sequential functions arguments of our skeletons and written in the host language (here C++) as Coq functions.

The result of a computation, a value, could be either a usual sequential value (for example the result of the application of the reduce skeleton) or a distributed array, for example the result of the application of the map skeleton.

For the library of skeletons, we do as usual when writing a formal semantics: we give the grammar of the language. Actually their are several ways of doing this. We shall illustrate this by two short examples dealing only with the construct for distributed arrays and the map skeleton. The first solution follows:

```
Inductive expression :=
| DistributedArray: ∀A:Type, list A →expression
| Map : ∀A B, (A→B) →expression →expression
```

To simplify the example, here a distributed array is just modelled as a list of values. All values being typed in Coq, the constructor for this case of the inductive type expression should also take as argument the type of the elements of the list. For the Map constructor, the first argument is the "muscle" argument, the function f to be applied to each element of the distributed array, the second expression. Here again the input and output types of the function should be given.

This grammar however models possibly ill-typed expressions of our language of skeletons. It is possible to define the following Coq term:

```
Definition e : expression := Map string string (append "!") (DistributedArray nat [1;2;3]).
```

In Coq it is possible to indicate than some arguments may be implicit: it is the case here for the types arguments of the two constructors Map and DistributedArray and we could write:

```
Definition e : expression := Map (append "!") (DistributedArray [1;2;3]).
```

The expression e is well typed for Coq but it *represents* an ill-typed expression of our skeleton language as the muscle function append operates on strings instead of naturals. We could as in [4] define a type system and prove that the operational semantics we will define follows the subject reduction property.

However there is another solution: we could model the grammar in such a way that only well-typed (in the skeleton language point of view) expressions could be modelled in Coq:

```
Inductive typedExpression (A:Type) :=
| TDistributedArray : list A →typedExpression A
| TMap: ∀B, (B→A) →typedExpression B →typedExpression A.
```

Here the grammar is typed. An expression of type typedExpression A represents an expression whose value is a distributed array whose elements have type A. The expression !e! could not be defined in Coq as a typedExpression as the input type of the muscle function in the Map constructor should be the type of the elements of the distributed array, second argument of Map.

If we could then defined the operational semantics by a function or a relation that relates only expressions that represent skeletons expression of the same type, then we would have the subject reduction for free.

The syntax of OSL is actually a bit more complicated as we distinguish between expressions whose values have a sequential type and expressions whose values have parallel types, these two kinds of expressions being mutually recursive. The whole syntax is in figure 2. In order to be able to apply a "sequential" programs to the result of the evaluation of a skeleton expression, we provide a SeqApply constructs. The SeqValue constructors is simply used to provide "muscles" to the skeletons.

The three first Coq constructors of the parExpr type are the usual OSL C++ class constructors: we can build a distributed array by specifying its size and a value that will be replicated everywhere, or the content of the distributed array could be specified by a function from array indices to values. In these two cases, the data is distributed evenly on the processors. The third constructor is used to build a distributed array containing values only at the root processor. The other Coq constructors model the skeleton informally presented in section 2.

4.3 Big-Step Semantics

For the formalisation of the big-step semantics of OSL, we define three functions, the two first begin mutually recursive:

- seqEvaluation: forall A : Type, seqExpr A → result A

```
Inductive seqExpr : Type →Type :=
| SeqValue: ∀A, A →seqExpr A
| Reduce: ∀A, seqExpr (A→A→A) →seqExpr A →parExpr A →seqExpr A
| SeqApply: ∀A B, seqExpr (A→B) →seqExpr A →seqExpr B
with parExpr : Type →Type :=
| DistributedArrayReplicate: ∀A, seqExpr A →seqExpr nat →parExpr A
| DistributedArrayInit: ∀A, seqExpr (nat→A) →seqExpr nat →parExpr A
| DistributedArrayCreateAtRoot: ∀A, seqExpr (list A) →parExpr A
| Map: ∀A B, seqExpr (A→B) →parExpr A →parExpr B
| Zip: ∀A B C, seqExpr (A→B→C) →parExpr A →parExpr B →parExpr C
| MapIndex: ∀A B, seqExpr (nat→A→B) →parExpr A →parExpr B
| Shift: ∀A, seqExpr Z →seqExpr(nat→A) →parExpr A →parExpr A
| GetPartition: ∀A, parExpr A →parExpr(list A)
| Flatten: ∀A, parExpr(list A) →parExpr A
| Permute: ∀A, seqExpr (nat→nat) →parExpr A →parExpr A.

Inductive expr : Type →Type :=
| Seq: ∀A, seqExpr A →expr A
| Par: ∀A, parExpr A →expr (distributedArray A).
```

Figure 2: OSL Syntax in Coq

- parEvaluation: forall A : Type, parExpr A → result (distributedArray A)
- evaluation: forall A : Type, expr A → result A

The result type is used in a monadic style [18] in order to model possible errors during evaluation, without being too cumbersome to use compared to a solution with optional values and pattern-matching. As in [12] for example, we use a convenient Coq feature that allows to define notations:

```
Inductive result (A: Type) : Type :=
| Ok: A →result A
| Error: string →result A.

Definition bind (A B: Type) (f: result A) (g: A →result B) : result B :=
  match f with
  | Ok x ⇒g x
  | Error msg ⇒Error msg
  end.

Notation "'do' X ␣<−␣ A ␣;␣ B" := (bind A (fun X ⇒B)).
```

With this notation, the big-step semantics functions are quite readable. For example the case for the evaluation of the reduce skeleton in the seqEvaluation function is written as follows:

```
| Reduce A op neutral pe ⇒
    do op <− (seqEvaluation op);
      do neutral <− seqEvaluation neutral;
```

```
    do da <- parEvaluation pe ;
      Ok(List.fold_right op neutral (distributedArray_data da))
```

We first evaluate the "muscles" of the skeletons. If one of these calls raises an error, then the function immediately returns this error, otherwise it binds the obtained value with the variable before the <− arrow and continues to evaluate the expression after the ;.

The parEvaluation function produces values of type distributedArray. In order to keep this function short, we defined auxiliary functions that transforms distributed arrays. The parEvaluation function thus first recursively calls itself and seqEvaluation on the arguments of the expression it evaluates, and obtains *values*, in particular in the parallel case, values of type distributedArray. Then it calls the appropriate auxiliary function. For example:

```
| DistributedArrayReplicate _ se se' ⇒
    do v <− seqEvaluation se;
      do size <− seqEvaluation se';
        Ok (replicate v size)
```

The replicate function, and all the auxiliary functions, are defined using the Program feature of Coq:

```
Program Definition replicate(A:Type)(value:A)(size:nat) : distributedArray A :=
  mkDistributedArray
  (List.map (fun index⇒value) (List.seq 0 (if beq_nat Bsp.p 0 then 0 else size)))
  (evenDistribution size)
  _.
Next Obligation.
  autorewrite with length; rewrite globalSizeEvenDistribution; trivial.
Defined.
```

For building a value of type distributedArray, we need three components:

- the content of the distributed array, in this case it is defined on the third line (we apply a constant function to all the elements of a list of naturals, of the specified size),
- the distribution, in this case it is defined on the fourth line, by a call to the function evenDistribution,
- a *proof* that the content and the distribution are coherent.

The two first components are written very similarly to functional programs. For the proof however, it is easier to use the interactive proof mode. Thus we do not give this third component: we use the wildcard _ instead. Coq then generates proof obligations that should be proved in order for the value replicate to be defined. The proof is here quite simple because most of the work is done in the lemma globalSizeEvenDistribution that it itself proved using several other lemmas.

This replicate function could not directly raise an error. Few skeletons can: the zip skeleton if the two parallel arguments do not have the same distribution and the flatten skeleton if the distribution of its argument is not one element (of type list) per processor.

Begin functions, evaluation, seqEvaluation and parEvaluation can be executed on OSL programs examples in Coq. By construction the type of the expressions are preserved during evaluation: we have subject reduction for free.

All the Coq source code of this formalisation is available at
`http://traclifo.univ-orleans.fr/OSL`.

5 Conclusion and Future Work

In this paper we have presented a formal semantics of the programming model of the Orléans Skeleton Library, modelled using the Coq proof assistant, also used to prove the properties of this semantics. It is a first step: we plan to design and implement in Coq formal semantics of the execution model and prove the equivalence with respect to the programming model. Writing such a formal semantics and checking its properties using a proof assistant make necessary to look into all the details of the semantics. Based on this semantics we improved the reliability of the current implementation of the OSL library in C++.

One limitation of this approach is that we are modelling the programs rather than trying to prove directly the code. This this mainly due to the fact that C++ is a complex programming language and, to our knowledge, there is no support for the proof of correctness of C++ programs with theorem provers or other tools. However to fill the gap between what is modelled and what is proved correct, we plan (it is a long term project) in the PaPDAS project (`http://traclifo.univ-orleans.fr/PaPDAS`) to design a skeletal parallel programming language, extension of C (not C++), and implement and prove correct a compiler for this language, building on the CompCert compiler [12].

References

[1] M. Aldinucci and M. Danelutto. Skeleton-based parallel programming: Functional and parallel semantics in a single shot. *Computer Languages, Systems and Structures*, 33(3-4):179–192, 2007.

[2] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.

[3] W. Bousdira, L. Gesbert, and F. Loulergue. Syntaxe et sémantique de Revised Bulk Synchronous Parallel ML. In S. Conchon and A. Mahboubi, editors, *Journées Francophones des Langages Applicatifs (JFLA)*, Studia Informatica Universalis, pages 117–146. Hermann, 2011.

[4] D. Caromel, L. Henrio, and M. Leyton. Type safe algorithmic skeletons. In *16th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 45–53. IEEE Computer Society, 2008.

[5] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989. Available at `http://homepages.inf.ed.ac.uk/mic/Pubs`.

[6] R. Di Cosmo, S. Pelagatti, and Z. Li. A calculus for parallel computations over multidimensional dense arrays. *Computer Language Structures and Systems*, 33(3-4):82–110, 2007.

[7] F. Gava and F. Loulergue. A Polymorphic Type System for Bulk Synchronous Parallel ML. In V. Malyshkin, editor, *Seventh International Conference on Parallel Computing Technologies (PaCT 2003)*, number 2763 in LNCS, pages 215–229. Springer Verlag, 2003.

[8] L. Gesbert, Z. Hu, F. Loulergue, K. Matsuzaki, and J. Tesson. Systematic Development of Correct Bulk Synchronous Parallel Programs. In *The 11th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 334–340. IEEE Computer Society, 2010.

[9] H. González-Vélez and M. Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software, Practrice & Experience*, 40(12):1135–1160, 2010.

[10] M. Hidalgo-Herrero and Y. Ortega-Mallén. An Operational Semantics for the Parallel Language Eden. *Parallel Processing Letters*, 12(2):211–228, 2002.

[11] N. Javed and F. Loulergue. OSL: Optimized Bulk Synchronous Parallel Skeletons on Distributed Arrays. In Y. Don, R. Gruber, and J. Joller, editors, *8th international Conference on Advanced Parallel Processing Technologies (APPT'09)*, LNCS 5737, pages 436–451. Springer, 2009.

[12] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[13] M. Leyton, L. Henrio, and J. M. Piquer. Exceptions for algorithmic skeletons. In P. D'Ambra, M. R. Guarracino, and D. Talia, editors, *16th International Euro-Par Conference*, LNCS 6272, pages 14–25. Springer, 2010.

[14] S. Pelagatti. *Structured Development of Parallel Programs*. Taylor & Francis, 1998.

[15] F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003.

[16] The Coq Development Team. The Coq Proof Assistant. `http://coq.inria.fr`.

[17] L. G. Valiant. A bridging model for parallel computation. *Comm. of the ACM*, 33(8):103, 1990.

[18] P. Wadler. Monads for Functional Programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, LNCS 925, pages 24–52. Springer, 1995.

A A Short Introduction to Coq

The Coq proof assistant [16] is based on the calculus on inductive construction. This calculus is a higher-order typed λ-calculus. Theorems are types and their proofs are terms of the calculus. The Coq systems helps the user to build the proof terms and offers a language of tactics to do so.

We illustrate quickly all these notions on a short example :

```
Inductive nat : Set :=
  | O : nat
  | S : nat →nat.

Fixpoint plus (n1 n2:nat)
  {struct n1} : nat :=
match n1 with
  | O ⇒n2
  | S n ⇒S(plus n n2)
end.

Lemma plus_n_O : ∀n, plus n O = n.
induction n.
 (* case n=0 *) simpl. reflexivity.
 (* case n>0 *) simpl. rewrite IHn. reflexivity.
Qed.

Definition pred :
 ∀n:nat, n<>O→{q:nat|(S q)=n}.
  intros.
  destruct n.
   (* case n=0 *) elim H. reflexivity.
   (* case n>0 *) exists n. reflexivity.
Defined.
```

In this example, we first define a new inductive type, the type of natural numbers in the Peano style. nat has type **Set** which means it belongs the computational realm of the Coq language. We also define the plus recursive function on naturals. In this recursive definition we specify the decreasing argument (here n1) as all functions must be terminating in Coq. In both cases, we gave the type of the new name we wanted to define as well as a term of this type.

We then define a lemma named plus_n_O which states that ∀n, plus n O = n. If we check (using the Check command of Coq) the type of expression, we would obtain Prop which mean that this expression belongs to the logical realm. To define plus_n_O we also should provide a term of this type, that is a proof of this lemma. We could write directly such a term, but it is usually complicated and Coq provides a language of tactics to help the user to build a proof term. If we give to Coq top-level the line beginning with **Lemma** we would enter the interactive proof mode that would indicate us that we should prove the goal:

```
  ============================
   forall n : nat, plus n O = n
```

We prove this goal by induction on n using the tactic induction n. The system indicates now two goals to prove:

```
  ============================
   plus O O = O

subgoal 2 is:
 plus (S n) O = S n
```

The first one is proved using the definition of plus using the tactic simpl which yields the goal 0 = 0 and this case is ended by the application of the tactic reflexivity. The second one is the inductive case:

```
  n : nat
  IHn : plus n O = n
  ============================
   plus (S n) O = S n
```

After simplification, we obtain the goal S(plus n O) = S n. We solve it first by rewriting plus n O in n using the IHn hypothesis and then we conclude by reflexivity.

Mixing logical and computational parts is possible in Coq. For example a function of type A→B with a precondition P and a postcondition Q corresponds to a constructive proof of type: ∀x:A, (P x) →exists y:B →(Q x y). This could be express in Coq using the inductive type sig:

Inductive sig (A:**Set**) (Q:A→Prop) : **Set** := | exist: ∀(x:A), (Q x) →(sig A Q).

It could also be written, using syntactic sugar, as {x:A|(P x)}.

This feature is used in definition of the function pred. The specification of this function is: ∀n:nat, n<>O→{q:nat|(S q)=n} and we build it using tactics. We reason by case on n (tactic destruct). The first case is easily solved because we have the hypothesis O<>O, the second one is trivial.