LEGO Proof Development System: User's Manual

Zhaohui Luo   Robert Pollack
Department of Computer Science
University of Edinburgh

May 7, 1992

Contents

1 Introduction

LEGO is an interactive proof development system (proof checker) designed and implemented by R. Pollack in Edinburgh. It implements various related type systems — the Edinburgh Logical Framework [HHP87], the Calculus of Constructions [CH88], the Generalized Calculus of Constructions [Coq86] and the Extended Calculus of Constructions [Luo89,90]. An overview of the basic theory and implementation of LEGO may be found in [Pol88], where most of the features of the system are sketched and explained, and a simple and informal introduction to the system can be found in [Bur89b]. In Pollack's forthcoming PhD thesis [Pol92], one may find much more detailed development on the system with a lot of interesting examples. As LEGO is still in the process of development, this document only describes the basic features of the current system so that it can be used by more people who are interested in theorem proving and program specification and development.

LEGO is a powerful tool for interactive proof development in the natural deduction style. It supports *refinement proof* as a basic operation. The system design emphasizes removing the more tedious aspects of interactive proofs. For example, the features of the system like argument synthesis [Pol90][CH88] and universe polymorphism [Pol90][HP91][Hue87] make proof checking more practical by bringing the level of formalization closer to that of informal mathematics. The higher-order power of its underlying type theories [CH88][Luo89,90], plus the support of specifying new inductive types, provides an expressive language for formalization of mathematical problems and program specification and development. Particularly, the type universes in the type theory make it possible to formalize abstract mathematics, and the strong sum types (Σ-types) can be used to naturally express abstract structures, mathematical theories [Luo91a][LPT89] and program specifications [Bur89a][BM91][Luo91b][McK92]. LEGO may also be used to formalize different logical systems and prove theorems based on the defined logics (*c.f.*, [HHP87]).

LEGO has been extensively used by many people and a lot of examples in theorem proving and program development have been done using it. These include a proof of the strong normalization theorem of the second-order λ-calculus [Ber90], the proof of Tarski's fixpoint theorem [Pol92], a formalization of generalized type systems [Pol92], development of the constructive real numbers from the rationals and the completion of a metric space [Jon91], a proof of Chinese Remainder Theorem [McK92], program specifications and program correctness proofs

[BM91][Luo91b][Hof91][McK92], development of a type synthesis algorithm for the simply typed λ-calculus (Pollack), formalizations of notions of sets [Mah91][Col91], a proof of Schroëdre-Bernstein Theorem (Hofmann), a proof of an isomorphism theorem for universal abstract algebra (Fairtlough), specification and implementation of the Odd-Even Transposition Sorter (Fairtlough, in progress), and proofs of the binomial theorem for rings and a bisimulation equivalence [Tay88,89].

Although we do not assume the reader is familiar with type theories, we do not explain basic type-theoretic notions. Instead, we try to give some intuitive analogies to help its understanding. LEGO users interact with the top level of ML [HMM86]. They are not required to be familiar with ML language, although some basic knowledge about ML would be useful. We do not assume that the reader of this manual has knowledge about ML.

Section 2 describes briefly the working environment to use LEGO. A simple example of refinement proof is given in section 3. Section 4 describes the top-level syntax of LEGO — the categories of expressions and their evaluations. The basic expressions called *terms* are described in section 5. Section 6 describes *contexts* which consist of declarations and definitions. Section 7 describes the special kind of LEGO expressions called *commands* and how they are used to develop proofs by refinement and manipulate contexts. Section 8 introduces the new feature for specifying inductive data types and explains its use by examples. Section 9 consists of examples of proof development and program specification and development. The syntax of LEGO expressions is given in Appendix A. Appendix B describes a logic file in which some basic logical definitions are given. Appendix C explains some ML functions one may use to work with LEGO on the top level of ML.

This document is a modified version of the LEGO user manual [LPT89]. Besides the explanation of some new commands and notational changes, and some new examples, the major extension is section 8, where the new feature of specifying inductive data types is explained, and section 9, where some new examples and the specification mechanism are described. The reader should note that the type theories on which LEGO is based are not formally discussed in this document; we refer the reader to [CH88][Luo90,91a] for formal definitions of the notions like conversion, well-typedness and principal types. We will take them for granted and use them in an intuitive (and precise) way.

The Lego group in Edinburgh, led by Rod Burstall, has had the following people as members who have contributed to the Lego project in various aspects: Thorsten Altenkirch, Jason Brown, Rod Burstall, Andy Coleman, Matt

Fairtlough, Healfdene Goguen, Martin Hofmann, Claire Jones, Zhaohui Luo, James McKinna, Savi Maharaj, Michael Mendler, Makoto Takeyama, and Paul Taylor. The people in the group have all benefited very much from the help of the secretarial work, especially by Eleanor Kerse.

Lego is available by anonymous ftp. You can get the system as follows:

```
ftp ftp.dcs.ed.ac.uk
Name: anonymous
Password: enter your e-mail address
cd export/lego
get README
```

Then, please read the file `README`.

Acknowledgements Z. Luo would like to thank Rod Burstall for encouraging him to write this document and giving many good suggestions, and give special thanks to Paul Taylor for his substantial contribution to the preliminary version of the manual. Thanks also go to Matt Kaufmann and many others for their comments and suggestions on the preliminary version of this manual, and to Claire Jones, who has kindly read the draft of this document and given helpful suggestions. Thanks to Martin Hofmann for contributing his example in section 9.5. R. Pollack wants to thank all LEGO users, especially the LEGO club at LFCS, and thank Th. Coquand, G. Huet, and Z. Luo for many conversations from which he has learned much about type theory and implementation.

2 The Working Environment

The working environment of LEGO is briefly described in this section.

2.1 Loading and exiting LEGO

The current LEGO uses Standard ML [HHM86] as a metalanguage. The LEGO user interacts with the top level of ML. To use LEGO system, (assuming that LEGO system is properly installed,) you start a Unix shell (maybe in an EMACS window) and load LEGO as follows:

```
shell% lego
Standalone LEGO
Generated 12-3
using Standard ML of New Jersey, Version 75, November 11, 1991
use command 'Help' for info on new commands.
Lego>
```

where '`Lego>`' is the prompt for LEGO.

'Ctrl-D' (`^D`) will enable you to exit lego state to return to UNIX.

```
Lego> ^D
shell%
```

2.2 Lego state and proof state

After loading LEGO (as shown above), the system enters a state called *lego state.* Another state is *proof state* which is entered from lego state when a new refinement proof starts, *i.e.*, when a `Goal` command is successfully evaluated (see section 7.2.1). In a proof state, there is a list of *current goals* to be solved; they are ordered with the first called *the current goal.* When the current goals are displayed (*e.g.*, using command `Prf`), the first one is the current goal.

The system enters a lego state from proof state when a refinement proof finishes (see 2.5).

The following picture may help you understand the general picture of LEGO system:

$$UNIX\ shell \xrightarrow{\texttt{lego}} lego\ state \rightleftharpoons proof\ \ state$$

One can talk to LEGO by *evaluating LEGO expressions* (see 4.2).

2.3 Initialization of different type systems

LEGO supports several basic type systems, which include

- LF: Edinburgh Logical Framework [HHP87]
- PCC: Pure Calculus of Constructions [CH88]
- CC: (Generalized) Calculus of Constructions [Coq86]
- XCC: Extended Calculus of Constructions [Luo89,90]

Roughly speaking, as pure formal systems, the above systems are related in the following order, from weaker to stronger: `LF`, `PCC`, `CC`, `XCC`. For details of the type theories and their relationships and differences, see [Pol88] and the papers referred above. The user can extend the type theories by specifying new inductive types (see section 8).

You can choose one of the above type systems as your basic type theory in which theorems are formulated (formalized) and proofs are developed. This is done using the command `Init` (initialization, see section 7.3.1), which takes one of the identifiers `LF`, `PCC`, `CC` or `XCC` as its argument to select the corresponding type theory and initializes the current context as empty (see section 6 for context). For example,

```
Lego> Init XCC ;
Extended CC: Initial State!
```

In the current delivered version of the system, when LEGO is loaded, `XCC` is automatedly initialized as a default.

2.4 Current context

The LEGO system maintains an environment, called the *current context*. The notion of context is inherited from the type theory. As explained above, each time a type theory is initialized, the current context is initialized to the empty one. All work after entering LEGO is done relative to current context. Evaluations of expressions (*e.g.*, execution of commands) may change the current context. You can also manipulate the current context using the commands for context manipulation (section 7.3).

2.5 Starting and finishing a proof

To start to prove a new theorem, one executes a `Goal` command (see 7.2.1) to conjecture a top-level *goal* (and enter a proof state).

A proof development can finish either successfully — `*** QED ***` is displayed (see 7.2.3), or by evaluating a `KillRef` command in a proof state (see 7.2.12)). When the top-level goal is successfully proved, the system automatedly discharges (see 7.3.5) all of the local assumptions and definitions done since the top-level goal was conjectured and enters lego state. Using `KillRef` aborts the current refinement and also enters a lego state.

3 An Example of Refinement Proof

This section gives a very simple example which explains the basic ideas of how to prove theorems by refinement in LEGO. More advanced and bigger examples may be found in section 9 and [Tay88][Tay89].

For the intuitive meanings of the syntax in the following example, we refer to section 5.2.

Example According to Curry-Howard correspondence, propositions of the calculus of constructions (the types of type `Prop`) can be viewed as logical formulas in its embedded intuitionistic higher-order logic. We prove in this simple example that one of the corresponding and-elimination rules (the left one) in the ordinary logic is admissible in the embedded logic. (*c.f.*, section 9.1.)

First, we introduce by definition (see section 6.2) a constant `and` which is defined in the way that AND operator is defined in higher-order logic:

```
Lego> and [A,B:Prop] = {C:Prop}(A->B->C)->C ;
defn  and = [A,B:Prop]{C:Prop}(A->B->C)->C
      and : Prop->Prop->Prop
```

The type of `and` (`Prop->Prop->Prop`) shows that, when given two propositions `A` and `B`, `and A B` is a proposition. We show that the proposition `and A B` implies the proposition `A`.

We use command `Goal` to start a new refinement proof by making a conjecture into a refinement goal which intuitively means that 'for all propositions `A` and `B`, if there is a proof of `and A B`, then there is a proof of `A`':

```
Lego> Goal {A,B:Prop}(and A B)->A ;
Goal
  ?0 : {A,B:Prop}(and A B)->A
```

The response of the LEGO system shows our conjecture is the current goal, which is numbered `0`. You may alternatively think of `?0` as a meta-variable, and it is a totally uninstantiated term whose type is the conjecture. Finding an instantiation of `?0` with the conjecture as type (*i.e.*, a proof of the conjecture) is the formal meaning of 'proving the conjecture'.

The first step of our refinement proof is to do some Π-introductions (*c.f.*, ∀-introductions in ordinary logics) which generate a subgoal in a context containing the hypotheses of the conjecture:

```
Lego> Intros A B ;
Intros A B
  A : Prop
  B : Prop
  ?1 : (and A B)->A
```

We now expand the definition of `and` in the current goal:

```
Lego> Expand and ;
Expand and
  ?1 : ({C:Prop}(A->B->C)->C)->A
```

and to solve this new goal, we do Π-introduction again (note that `C->D` is an abbreviation form of `{x:C}D` when there is no dependency of `D` on `x`):

```
Lego> Intros h ;
Intros h
  h : {C:Prop}(A->B->C)->C
  ?2 : A
```

Now, in order to get a proof of `A`, we try to use the proof of `and A B` (`h`) to proceed a refinement step:

```
Lego> Refine h ;
Refine by  h
  ?4 : A->B->A
```

Here is what happened in the above refinement step. The LEGO system tried to unify the (principal) type of the refinement term `h` (*i.e.*, `and A B`) with the type of `?2` (*i.e.*, the current goal `A`). If it succeeds, the goal is completely solved by the refinement. In the above case, it fails. LEGO then tries to 'specialize' the refinement term by applying it to a new meta-variable of the right type and tries the new refinement; and continues until either unification succeeds, or the refinement fails because there are no more outer Π-binding operators on the refinement type. In this case, unification succeeds in the second step and resulted in a new goal as above.

We are now at a stage required to prove a most simple axiom in logic. We introduce again and do a refinement step which will lead us to the success of proving the conjecture:

```
Lego> Intros a b ;
Intros a b
  a : A
  b : B
  ?5 : A

Lego> Refine a ;
Refine by  a
Discharge..  b a
Discharge..  h
Discharge..  B A
*** QED ***
```

Finding no remaining goals in the current context, the system discharges (see section 7.3.5) the local assumptions. When finding no remaining goals in the discharged context and no further discharges to do, `*** QED ***` is printed which signals the success and end of the refinement proof. The system exits proof state and enters a lego state.

We have proved the admissibility of the (left) AND-elimination and can now save the actual proof term we get at the end of a refinement proof as a defined constant in the context:

```
Lego> Save and_elim_left ;
and_elim_left  saved
```

and examine its value and type:

```
Lego> and_elim_left ;
value = [P,Q:Prop][h:{X:Prop}(P->Q->X)->X]h P ([p:P][_:Q]p)
type  = {P,Q:Prop}(and P Q)->P
```

Being saved (that is, it is added to the current context as a constant denoting the proof) `and_elim_left` can be used as a lemma in later proof development.

4 Expressions

We describe in this section the top-level syntax of LEGO — *expressions* and their evaluations.

4.1 Categories of expressions

The syntactic expressions of LEGO may be classified into the following three main syntactic categories:

1. Terms (see section 5)
2. Contexts (declarations and definitions) (see section 6)
3. Commands (see section 7)

Terms and contexts are basically those inherited from type theory, and commands are those mainly for proof development and context manipulation which will be described in section 7. The complete syntax of LEGO expressions can be found in Appendix A and consulted when necessary. The top-level syntax of LEGO expressions can be described by the following BNF grammar:

```
exps ::= exp | exps ; exp
exp  ::= term | context | command | definition
```

A sequence of expressions can be formed by concatenating expressions by semicolon (;) as the above grammar shows.

Remark At present, as the above grammar shows, some syntactic sugars of definitional context components (the category `definition`, see 6.2) are also expressions.

4.2 Evaluation of (sequences of) expressions

An expression (or a sequence of expressions) can be *evaluated* (in lego state or proof state). A semicolon (and 'return') will cause the expression to be evaluated:

```
Lego> exp ;
```

This is the way you interact with the LEGO system.

The evaluation of an expression can either *succeed* or *fail*. A sequence of expressions are evaluated in order (one by one) and, whenever the evaluation of one of them fails, the rest of them will not be evaluated but we still gain the results and effects of the evaluations of the expressions before it. We say the evaluation of a sequence of expressions *succeeds* if, and only if, the subsequent evaluations of the expressions in the sequence all succeed. Otherwise, we say it *fails*.

In general, evaluating an expression changes the state of the system. Evaluating different categories of expressions will get different results and effects. Evaluating a command is also called executing it. These will be explained when different categories of expressions are introduced.

5 Terms

Terms are the basic expressions in LEGO. They are essentially the LEGO representations of the basic expressions in the underlying type theory with some generalizations like existential variables. We first give a general description of the syntax of terms and then explain their intuitive meanings.

5.1 Syntax of terms

A complete syntax for terms can be found in Appendix A. Instead of directly interpreting the syntax in the appendix, we give a simpler grammar, where `M` ranges over terms, `x` over identifiers and `n` over natural numbers.

```
M ::= x | Prop | Type | Type(n) |
      {x:M}M | [x:M]M | M M    |
      M->M   | M\M    | M.M    |
      {x|M}M | [x|M]M | M|M    |
      <x:M>M | M,M    | M.1 | M.2 | M#M |
      [x=M]M | (M) | M:M |
      ? | ?n | ?+n
```

The terms except those containing subterms of the form in the last line of the above grammar are basically those inherited from the underlying type theory. Not all of the them exist in all of the type theories LEGO supports, although they do in the 'most general' one (Extended Calculus of Constructions). The intuitive meanings of the terms are explained in 5.2.

Note that the grammar we give is ambiguous. Some syntactic conventions are described below.

5.1.1 scopes and associativity

The *scopes* of the binding operators `{x:M}`, `[x:M]`, `{x|M}`, `[x|M]`, `<x:M>` and `[x=M]` are determined by the rule that they extend *as far to the right as possible*; in other words, their scopes stops by the first colon (`:`) or right parenthesis at the same parsing level. The scope of the operator `:` used to form a term of the form `M:M` expands to both directions as far as possible. For example, the following terms

```
{x:M}M1->(M2:Type(0)):Type(1)
([x:M]M1 M2) M3
(<x:M>M1->M2)->M3
```

stand respectively for

```
({x:M}(M1->(M2:Type(0)))):M3
([x:M](M1 M2)) M3
(<x:M>(M1->M2))->M3
```

and the scopes of `{x:M}`, `[x:M]` and `<x:M>` all extend to `M1` and `M2` but not `M3`.

The scopes of the operators `->`, `\`, `#` and `,` (comma in `M,M`) also extend as far to the right as possible, but they have *smallest scope to their left-hand side*. For example, the following terms

```
A#B->C#D      f a,b,c
```

stand respectively for

```
A#(B->(C#D))      f (a,(b,c))
```

The operators `:` (for type-casting `M:M`), `.` (dot) and the application operators (the space in `M M` and the bar in `M|M`) are *left associative*. For example,

```
M1:M2:M3   M1.M2.M3     M.1.2.1     M1|M2|M3
```

stand respectively for

```
(M1:M2):M3   (M1.M2).M3     ((M.1).2).1     (M1|M2)|M3
```

Furthermore, in LEGO terms with one of the following forms:

```
M->N   M#N   M.1   M.2   M.M
```

`M` should not be thought of as being of an application form like `M1 M2` or `M1|M2` without parentheses surrounded. In other words, for example, `M1 M2->N` stands for `M1 (M2->N)` instead of `(M1 M2)->N`. The parentheses are essential here.

5.1.2 some abbreviations

In LEGO, the following terms

```
{x1:A}...{xn:A}B   [x1:A]...[xn:A]M   <x1:A>...<xn:A>B   [x1=A]...[xn=A]B
```

can be abbreviated by the following simpler terms,

```
{x1,...,xn:A}B     [x1,...,xn:A]M     <x1,...,xn:A>B     [x1,...,xn=A]B
```

respectively. Furthermore, the terms

```
A->B      A#B      A\B
```

can be abbreviated respectively as

```
{_:A}B   <_:A>B    [_:A]B
```

So, for instance, `{x,_,z:A}B` will stand for `{x:A}(A->{z:A}B)`.

5.2 Terms and their intuitive meanings

We explain the intuitive meanings of the terms in this section.

5.2.1 identifiers

`x` stands for an arbitrary variable (for both declarations and definitions) which can be any identifier (*i.e.*, of type `IDENT` in ML).

5.2.2 type universes

`Prop`, `Type(n)` (`n` ≥ 0 stands for a natural number), and `Type` are called *type universes*, which constitute a hierarchy like set theoretic universes. Intuitively (and semantically), one may think of types as sets. In general, any term which has a universe as its type is called a *type*. When a term has type `Prop`, it is also called a *proposition*.

The type universes are also types themselves. We have

$$\texttt{Prop} : \texttt{Type(0)} : \texttt{Type(1)} : ...$$

That is, `Prop` is of type `Type(0)`, `Type(0)` is of type `Type(1)`, ... *etc.*. This is the basis of the strong polymorphism in the type theory.

Furthermore, we have the following type inclusions for the universes:

$$\texttt{Prop} \subseteq \texttt{Type(0)} \subseteq \texttt{Type(1)} \subseteq ...$$

That is, every term which is of type `Type(i)` is also of type `Type(j)` for any `j` $\geq$ `i`. And, every proposition is of type `Type(0)`.

LEGO supports universe polymorphism (or typical ambiguity) [HP91][Hue87]. One can omit the subscripts of `Type(n)` and work with LEGO just as if `Type` were the type of all types and were the type of itself.

5.2.3 Π-types and λ-abstraction

`{x:A}B` is the LEGO notation for ordinary dependent Π-types ($\Pi x{:}A.B$) in type theory. `{x:A}B` is a well-typed type in a context C if, and only if, `A` is a well-typed type in C and `B` is a well-typed type in C,`x:A`. It is a proposition in C if, and only if, `A` is a well-typed type in C and `B` is a proposition in C,`x:A`. `{x:A}B` is often writen as `A->B` when the identifier `x` does not occur free in `B`. When `{x:A}B` is a proposition, it intuitively stands for the logical formula '$\forall x{:}A.B$'. When it is not a proposition, it intuitively denotes a set of functions f with domain `A` such that f(`a`) is of type `[a/x]B`.[1] Particularly, if both `A` and `B` are propositions, the proposition `A->B` intuitively means '`A` implies `B`'; if neither `A` nor `B` is a proposition, `A->B` intuitively stands for the function type from `A` to `B`.

`[x:A]M` is the LEGO notation for ordinary λ-form ($\lambda x{:}A.M$) which intuitively denotes a function which, whenever applied to an element a of type `A`, returns the

[1][a/x]B is the term obtained by substituting all free occurrences of x by `a` in B.

value of `[`a`/x]M` as its value. You may use `A\M` to abbreviate `[x:A]M` when `x` does not occur free in `M`.

`M N` intuitively denotes the result of *function application* of the function denoted by `M` to the value denoted by `N`. You may use `N.M` to abbreviate `(M N)`.

In Constructions with universes, quantification over type universes is allowed; this provides a rather strong polymorphism. For example, the following function definition

```
compose [A,B,C|Type][f:A->B][g:B->C] = [x:A]g (f x)
```

would define a polymorphic function `compose` which, when given two functions `f` and `g` with appropriate types, returns their composition `compose f g` as result. Notice that in this example, `compose` has type

$$\{\mathtt{A,B,C|Type}\}\{\mathtt{f{:}A-}>\mathtt{B}\}\{\mathtt{g{:}B-}>\mathtt{C}\}\mathtt{A-}>\mathtt{C}$$

The presence of the vertical bar (`|`) means that the first three arguments of `compose` will be omitted when `compose` is applied. For instance, we can say `compose double suc` instead of writing `compose nat nat nat double suc`. The system will deduce the appropriate types from the forms of `f` and `g`. For functions defined using `|` like `compose` above, you can also indicate the intended arguments which are supposed to be omitted using '`|`-application'. For example, you can write `compose|nat|nat|nat double suc`. Correspondingly, we have terms of the form `[x|A]M` which has type `{x|A}B` if `[x:A]M` has type `{x:A}B`.

5.2.4 Σ-types, pairs and projections

`<x:A>B` is the LEGO notation for ordinary (dependent) strong sum types (Σ-types, $\Sigma x{:}A.B$). `<x:A>B` is a well-typed type in a context C if, and only if, `A` and `B` are well-typed types under C and C, `x:A`, respectively. `<x:A>B` intuitively denotes the set of pairs (a, b) such that a is in the set denoted by `A` and b is in that denoted by `B(`a`)`. `A#B` is the abbreviation of `<x:A>B` when `x` does not occur free in `B` which denotes the usual product type ($A \times B$ in the usual notation).

`a,b` is the form of a pair in LEGO which is formed by operator `,`. The principal type of `a,b` is `A#B` where `A` and `B` are the principal types of `a` and `b`, respectively. One can use type-casting to assure that a pair has intended dependent Σ-type as its principal type. (See 5.2.6.)

`M.1` and `M.2` are the two projections (in the usual notation $\pi_1(\mathrm{M})$ and $\pi_2(M)$) which intuitively extract the first and second components of `M` respectively.

Strong sum types can be used to express abstract structures and mathematical theories (see [Luo91a][LPT89]) and specifications of abstract data types (see section 9.3).

5.2.5 local definitions

`[x=M]N` is the LEGO notation for local definitions (*c.f.*, the let-construct *let* $x =$ M *in* N *end* in functional languages). Roughly speaking, it is equivalent to the term `[M/x]N`. Because of the presence of universe polymorphism, it behaves like the let-construct in ML.

5.2.6 type-castings

Every (well-typed) term `M` can be cast with one of its types `A` to form a new (well-typed) term `M:A`. Intuitively, `M:A` is just the term `M` except that its principal type is `A` which may be different from the principal type of `M`.

For example, let `TRUE` and `true` denote the following terms

```
TRUE = {P:Prop}P->P
true = [P:Prop][p:P]p
```

Then, the term

```
(TRUE, true)
```

will have `Prop#TRUE` as its principal type and does *not* have type `<x:A>x`. But the term

```
(TRUE,true):<x:Prop>x
```

has `<x:Prop>x` as its principal type. It does not have type `Prop#TRUE`.

Note that, when `M` has type `A` in context with `x:B`, the term `[x:B](M:A)` is well-typed and it is equivalent to the term `[x:B]M:{x:B}A`.

5.2.7 existential variables

`?n`, `?+n` (with `n` ≥ 0 being a natural number) and `?` are called *existential variables*.

`?n` (`?+n`) can only be used in a proof state when a goal numbered `n` (the (n+1)th current goal) exists. `?n` (`?+n`) stands for the uninstantiated proof of the goal numbered by `n` (the (n+1)th current goal), which is its principal type. It can be used to form well-typed terms, just as any other term.

For example, at some stage of a proof development, there may be a situation that there are two goals which are the same (say `A`):

```
  ?1 : A
  ?2 : A
```

Then, we can use `?2` as a refinement term to solve the first goal (see section 7.2.3 for the refinement command):

```
 Refine ?2 ;
Refine by  ?2
  ?2 : A
```

In a more useful case, `?n` can be used together with the `Claim` command to postpone proving a lemma.

`?` is more subtle. It can be used either in lego state or proof state. It can be used to form terms just as any other terms *except* that it cannot occur as the function part of an application (that is, `? M` is illegal). The system uses unification to try to figure out the real term that `?` stands for. In a lego state, if the unification fails, the term containing `?` is not well-typed. In a proof state, if the unification fails, a new goal is generated which asks the user to provide the incomplete part of the real term.

`?` can be rather helpful when proving bigger theorems. For example, instead of having defined the composition function `compose` as in section 5.2.3, we may have defined a function `Compose` without using the facility of argument synthesis:

```
Lego> Compose [A,B,C:Type][f:A->B][g:B->C] = [x:A] g (f x) ;
defn  Compose = [A,B,C:Type][f:A->B][g:B->C][x:A]g (f x)
      Compose : {A,B,C:Type}(A->B)->(B->C)->A->C
```

Suppose we have defined a (propositional) function `not` which is of type `Prop->Prop`. Then, we can't apply `Compose` to `not` directly:

```
Lego> Compose not not ;
attempt to apply  Compose
with domain type  Type
to  not : Prop->Prop
```

Instead, we can use the existential variable `?` to ask the system to synthesize the necessary arguments:

```
Lego>Compose ? ? ? not not ;
value = Compose Prop Prop Prop not not
type  = Prop->Prop
```

5.3 Evaluation of terms

Terms, as a special kind of expressions, can be evaluated (see section 4). Evaluation of a term gives as results its *value* (that is the term itself) and its (principal) *type* (in the current context). For example,

```
Lego> [P:Prop][p:P]p ;
value = [P:Prop][p:P]p
type  = {P:Prop}P->P
```

The system maintains the *current term* and its (principal) type. After evaluation of a term, the just evaluated term becomes the current one. The evaluation of a term only succeeds when the term is well-typed in the current context.

The current term and its type are saved in two registers called `VReg` and `TReg`, respectively. Some commands like `Equiv` and `Expand` can be used to make access to them.

6 Declarations, Definitions and Contexts

Contexts are sequences composed of entities called *declarations* and *definitions* which we will describe in the next two subsections (6.1 and 6.2).

Intuitively, a context declares a set of constants and variables with their types and postulates some axioms. New context components can be constructed by using those identifiers already declared or defined.

6.1 Declarations and their evaluation

A *declaration* is an expression of the following form:

[`x1`, ..., `xn`:`M`]

where `xi`'s are identifiers and `M` is any term which is not of the form `M1:M2` (type-casting).

A declaration can only be evaluated in a lego state. It is not allowed to evaluate a declaration in a proof state.

In a lego state, the evaluation of a declaration `[x:M]` succeeds if, and only if, `x` is a new identifier which does not occur free in the current context and `M` is a well-typed type in the current context; and if so, it has the effect that the evaluated declaration `x:M` is concatenated to the current context and the resulted context becomes the (new) current context. The evaluation of `x1,...,xn:M` succeeds if, and only if, the evaluations of the declarations `x1:M`, ..., `xn:M` in that order all succeed; and if so, they have the same effect.

For example, (after initializing `CC` or `XCC`,) we can declare a type `nat` (for natural numbers):

```
Lego> [nat:Type] ;
decl  nat :Type
```

which becomes the current context

```
Lego> Ctxt ;
  nat : Type
```

Then, we may further declare the natural number constructors by declaring (evaluating) the following declarations:

```
Lego> [zero:nat][s:nat->nat] ;
decl  zero :nat
decl  s :nat->nat
```

Then, the current context becomes

```
Lego> Ctxt ;
  nat : Type
  zero : nat
  s : nat->nat
```

6.2 Definitions and their evaluation

Introducing definitional extensions (to the current context) can be done in LEGO. δ-reduction is the basic mechanism of implementing definitions.

The formal syntactic form of a *definition* is of the following form:

$$[\texttt{c C} \;=\; \texttt{M : A}]$$

where `c` is an identifier, `C` is a context (see 6.3 below), `M` is a term and `A` is a term which is not of the form `A1:A2`. `: A` above is optional.

Definitions can be evaluated either in a lego state or in a proof state. The evaluation of a definition of the form `[c = M]` (when `C` is the empty context and `A` is not given) succeeds if, and only if, `c` is a new identifier which does not occur free in the current context and `M` is well-typed in the current context; and if so, the definition is concatenated to the current context and the resulted context becomes the (new) current context and the principal type of `c` is that of `M`.

Suppose `C` is of the form

`b1 b2 ... bn`

where `bi` is either a declaration or a definition. Then, evaluating `[c C = M : A]` is equivalent to evaluating the following definition:

[`c` = `b1 b2 ... bn (M:A)`]

In particular, since a term of the form `[x:B](M:A)` is equivalent to the term `[x:B]M:{x:B}A` (*c.f.*, 5.2.6), the evaluation of definition

[`c [x1:A1]...[xn:An]` = `M:A`]

has the same effects as that of

[`c` = `[x1:A1]...[xn:An]M : {x1:A1}...{xn:An}A`]

For example, continuing the example in the last subsection we can define a new function `plus2` as follows:

```
Lego> plus2 [x:nat] = s (s x) : nat ;
defn  plus2 = [x:nat]s (s x)
      plus2 : nat->nat
```

Then, the current context becomes:

```
Lego> Ctxt ;
  nat : Type
  zero : nat
  s : nat->nat
  plus2 = [x:nat]s (s x)
    : nat->nat
```

There are several forms of syntactic sugars for the definition `[c C = M : A]`, which are (`: A` is also optional)

```
c C = M : A
c C : A = M
```

6.3 Contexts and their evaluation

Contexts are (possibly empty) sequences of declarations and definitions, *i.e.*, they are of the form `b1 ... bn`, where `bi` is either a declaration or a definition. The variables declared before (say, that in `bi`) can be used to declare or define new variables later on (say, in `bj` for `j>i`). The evaluation of a context of this form succeeds if, and only if, the evaluation of the expression sequence `b1; ...; bn` succeeds; and if so, they have the same effect.

LEGO system maintains the *current context*. After the initialization of a type theory (see section 2.3), the current context is initialized as the empty one.

7 Commands

Commands constitute a special category of LEGO expressions (see section 4) which are mainly used in the process of proof development, in particular, to perform refinement proofs and to manipulate contexts.

7.1 Commands — general syntax

A command consists of a *command name* and several (possibly zero) arguments separated by one or more blanks between them:

$$C\ a_1\ ...\ a_n$$

Different commands may take different kinds of arguments (most of them are some kinds of expressions with a few exceptions).

7.2 Basic commands for refinement proofs

The commands used for refinement proofs are described in this section.

7.2.1 Goal

To start a new refinement proof under a lego state, we declare a new top level goal. The command `Goal` takes one term as its argument:

```
Goal M
```

which, when evaluated, makes a conjecture (the term `M`) be the top level goal (the current goal) to be proved. After `Goal M` is evaluated successfully, the system enters a proof state and a new refinement becomes alive.

Note that `Goal` can only be used in a lego state. If you want to start another new refinement proof during the process of a refinement proof (in a proof state), you can use `KillRef` (see 7.2.12) to kill the current refinement and then use `Goal` to start a new proof development.

7.2.2 Intros and intros

The commands `Intros` and `intros` are the commands for both Π-introduction and Σ-introduction. They can only be used in a proof state. They have the form

`Intros n` a_1 ... a_n

`intros n` a_1 ... a_n

where `n` is an (optional) natural number, $a_1, ..., a_n$ is a (possibly empty) sequence of `#`, `_` or different identifiers.

The evaluation of `Intros a` (where `a` is not `#` or `_`) succeeds if, and only if,

1. `a` is a new identifier not occurring in the current context, and
2. the current goal is convertible to the form `{x:A}B` (or `A->B`)

and if so, it has the following effects:

- the declaration `a:A` is concatenated to the current context, and
- a new goal `[a/x]B` is generated to replace the current goal and becomes the current goal.

`Intros _` is the same as `Intros a` (with `a` an identifier) except, when it succeeds, the system will generate a new identifier to play the role of `a` above.

The evaluation of `Intros #` succeeds if, and only if, the current goal is convertible to the form `<x:A>B` (or `A#B`); and if so, it has the effect that two new goals

```
?n : A
?n': [?n/x]B
```

are generated to replace the current goal and `?n : A` becomes the new current goal.

The evaluation of `Intros` a_1 ... a_n (`n` ≥ 1) succeeds if, and only if, the evaluation of the sequence of commands `Intros` a_1; ...; `Intros` a_n succeeds; and if so, they have the same effect. Evaluating `Intros` (with no argument) has the same effect as evaluating successfully a maximum number (possibly zero) of `Intros _` successively.

The evaluation of `Intros n` a_1 ... a_n succeeds if, and only if, the evaluation of the sequence of commands `Next n`; `Intros` a_1 ... a_n does; and if so, they have the same effects.

The command `intros n a1 ... an` is the same as `Intros n a1 ... an`, except that `intros a` succeeds only when the current goal *is* of the form `{x:A}B` or `A->B` (when `a` is not `#`) or of the form `<x:A>B` or `A#B` (when `a` is `#`). (For `Intros`, we only require that the current goal is *convertible* to such a form.)

7.2.3 Refine

The command `Refine` has the following three forms:

`Refine M` or `Refine n M` or `Refine +n M`

where `M` is a term and `n` is a natural number. It can only be used in a proof state.

The evaluation of a refinement command of the form `Refine M`, when `M` is well-typed in the current context, proceeds as follows: First, the system tries to unify the current goal with the principal type of the refinement term `M`. There are several possibilities:

1. the unification succeeds: then, the current goal is proved.
2. the unification fails: then, the system tries to specialize the refinement term `M` by applying it to a new metavariable of the right type. Two possibilities:
   - specialization succeeds: then, several new goals to be solved in order to prove the current goal are generated;
   - specialization fails: the refinement step fails.

Furthermore, when the system succeeds in solving the current goal by executing a `Refine` command and also find out that there is no further goal left, it discharges (see 7.3.5) all the local assumptions introduced by `Intros` at this level; If the top level goal is solved, `*** QED ***` is displayed and the system exits proof state and enters a lego state.

See section 3 for some explanations in the given example.

The evaluation of `Refine n M` (`Refine +n M`) succeeds if, and only if, the evaluation of the command sequence `Next n; Refine M` (`Next +n; Refine M`) does; and if so, they have the same effects. (See 7.2.5 for `Next`.)

7.2.4 Immed

The command `Immed` takes no argument. It can only be used in a proof state. Evaluating `Immed` will remove all of the goals that can be completely resolved by some entry in the current context (*i.e.*, the goals that unify with some types in the context). If the top-level goal is solved by `Immed` command, `*** QED ***` is displayed and the system exits proof state and enters a lego state.

See 9.1 for examples.

7.2.5 Next

`Next` takes `n` or `+n` as its argument, where `n` is a natural number. It can only be used in a proof state. Evaluating `Next n` will cause the goal numbered `n` (`?n : A`) to become the current goal; evaluating `Next +n` will cause the (`n`+1)th current goal to become the current goal.

For example, suppose we have the following goals to solve:

```
  ?1 : A
  ?8 : B#C
```

and we hope to solve the second one first. We can use `Next` to do change the order of the goals:

```
Lego> Next 8 ;
Next 8
```

Now, the goal numbered `8` becomes the current goal:

```
Lego> Prf ;
  ?8 : B#C
  ?1 : A
```

In the above example, instead of executing `Next 8`, evaluating `Next +1` has the same effect.

7.2.6 Expand

`Expand` takes a (non-empty) sequence of identifiers as its arguments. It is either of the form

```
Expand x1 ... xn
```

where `xi`'s are identifiers, or of the form

`Expand` R `x1 ... xn`

where R is either the register `VReg` for the current term or the register `TReg` for the type of the current term.

The former form of the `Expand` command can only be used in a proof state; the later form can be used either in a lego state or in a proof state.

The evaluation of `Expand x` (`Expand VReg x`, `Expand VReg x`, respectively) succeeds if, and only if, `x` is an identifier defined by a definition `x=M` in the current context; and if so, it has the effect that the current goal (the current term, the type of the current term) `A` is replaced by `[M/x]A`.

The evaluation of `Expand x1 ... xn` (`Expand R x1 ... xn`) succeeds if and only if, the evaluation of the sequence of commands `Expand x1 ; ... ; Expand xn` (`Expand R x1; ... Expand R xn`) does; and if so, they have the same effects.

See 9.1 for examples.

7.2.7 Equiv

`Equiv` command has the following two forms:

`Equiv M`

`Equiv R M`

where `M` is a term and `R` is either `VReg` or `TReg`.

The first form can only be used in a proof state. The later can be used either in a lego state or in a proof state.

The evaluation of `Equiv M` succeeds if, and only if, `M` is well-typed in the current context and convertible to the current goal; and if so, the current goal is replaced by `M`. For example, if we have the following goal to be proved:

```
  ?0 : ([x:Type]x) A
```

we can instead to prove an equivalent (convertible) goal `A`:

```
Lego> Equiv A ;
Equiv
  ?0 : A
```

The evaluation of `Equiv VReg M` (`Equiv TVeg M`) succeeds if, and only if, `M` is well-typed in the current context; and when it succeeds, it will answer whether `M` is equivalent (convertible) to the current term (the type of the current term) and, if it is the case, `M` becomes the current term (the type of the current term).

When used in a proof state (in the first form), this command corresponds to the type equality (conversion) rule in the type theory. It is important as first-order unification may fail to unify two unifiable higher-order terms.

7.2.8 Qrepl

The command `Qrepl` takes a term as its argument. It can only be used in a proof state (and for the type theory ECC, PCC or CC).

The evaluation of command `Qrepl M` will succeed under the following conditions:

1. in the current proof state, the term `M` has a type of the form `Q a b`, where `Q` is the Leibniz's equality (as defined in the logic file), and
2. the current goal `G` of the current proof state is of type `Prop`.

When the command succeeds, a new goal `G1` is generated as the next current goal. The new goal `G1` is the term resulted from replacing by the term `b` all of the subterms in the current goal `G` which are convertible to term `a`.

The command `Qrepl` may be 'configured' for any equality that is substitutive and symmetric. The command `Configure Qrepl` takes three identifiers as its arguments:

```
Configure Qrepl Eq Eq_subst Eq_sym
```

such that `Eq`, `Eq_subst`, and `Eq_sym` are the names of the equality type, its substitution operator, and its symmetry operator, which are required to be of type `{A|Type}A->A->Prop`, `{A|Type}{a,b|A}(Eq a b)->{P:A->Prop}(P a)->P b`, and `{A|Type}{a,b|A}(Eq a b)->Eq b a`, respectively.

After the above command is successfully evaluated, `Qrepl` will work for `Eq` instead of the Leibniz equality. You may use `Configure Qrepl` many times to change equalities for the `Qrepl` command. To re-establish the default Liebniz equality, use `Configure Qrepl` with no arguments.

7.2.9 Claim

`Claim` takes one term as its argument. It can only be used in a proof state.

Evaluating `Claim M` (with `M` well-typed term in the current context) will add `M` to the current goal list to become the last goal.

This command can be used to postpone proving some lemmas. (*c.f.*, section 5.2.7).

7.2.10 Prf

`Prf` takes no argument. It can only be used in a proof state.

Evaluating `Prf` will display the current goals (with the current goal displayed as the first) and context entries introduced in the current refinement proof (say, by using the command `Intros`).

7.2.11 Undo

`Undo` takes a natural number as argument.

The system maintains the refinement history after each time a top-level goal is conjectured (using a `Goal` command). Evaluating `Undo n` successfully will undo `n` steps of refinement proof process, returns back to an appropriate proof state, and forget all activities done after that.

After LEGO prints `***QED***`, it is still possible to use `Undo` to go back into the just-completed proof; however, the commands `Save`, `Forget` and `Discharge` will throw away the refinement history and their evaluations will disallow `Undo` to be used until the next refinement proof begins.

7.2.12 KillRef

`KillRef` takes no argument. It can only be used in a proof state. Evaluating `KillRef` will undo all actions done since the latest `Goal` command and causes the system to enter lego state.

7.3 Commands for context manipulation

Contexts were described in section 6. The commands to manipulate contexts are described in this section.

7.3.1 Init

For `Init`, see section 2.3 for its description. This command can only be used in a lego state.

7.3.2 Ctxt and Decls

`Ctxt` and `Decls` either take a natural number as its argument or take no argument. They can be used either in a lego state or in a proof state. Evaluating `Ctxt` (`Ctxt n`) will cause the whole current context (the latest `n` entries of the current context) to be displayed. Evaluating `Decls` (`Decls n`) will cause all (the latest `n`) declarations, but not the definitions, in the current context to be displayed.

7.3.3 Forget

`Forget` takes an identifier as its argument. This command can only be used in a lego state. The evaluation of `Forget c` succeeds if, and only if, `c` occurs in the current context; and if so, it has the effect that all the context entries back to and including the one named by `c` are removed.

For example, if the current context is as follows:

```
Lego> Ctxt ;
  and = [A,B:Prop]{C:Prop}(A->B->C)->C
    : Prop->Prop->Prop
  A : Type
  B : Type
  C : Type
```

After evaluating

```
Lego> Forget B ;
forgot back through B
```

the current context becomes

```
Lego> Ctxt ;
  and = [A,B:Prop]{C:Prop}(A->B->C)->C
    : Prop->Prop->Prop
  A : Type
```

7.3.4 Save

The command `Save` takes an identifier as argument and can be evaluated after a refinement proof is finished (with `QED` printed) and before a command `Save`, `Forget` or `Discharge` is evaluated. Evaluating `Save c` makes `c` be defined as the proof term with the proceeding goal as its type. This new definition is added to the current context so that the theorem just proved can be used as a lemma later on. See section 3 for an example.

7.3.5 Discharge and DischargeKeep

`Discharge` and `DischargeKeep` take an identifier as their argument. They can only be used in a lego state.

The evaluation of `Discharge x` succeeds if, and only if, `x` is in the current context; and if so, it 'discharges' and forgets all the identifier declarations back to and including the one named by `x`. `DischargeKeep x` is the same as `Discharge x` except that it does not forget the declarations. The word 'discharge' is informally explained in the following example.

Proofs may sometimes become rather tedious when they are cluttered by irrelevant polymorphism. Declarations and the command `Discharge` can help us to avoid such difficulties [CH85]. For example, instead of postulating that 'for arbitrary type `A`, for arbitrary binary relation `R` over the type, ... ', we can first declare an *arbitrary* type `A` and an *arbitrary* relation over `A`:

```
Lego>[A:Type][R:A->A->Prop];
decl  A :Type
decl  R :A->A->Prop
```

and then work in this context, for example, to declare new variables, to define new constants, to prove new theorems and save them, *etc.*. For instance, we can define the proposition that express the reflexivity property of `R` as follows:

```
Lego> Reflex = {x:A}R x x;
defn  Reflex = {x:A}(R x x)
      Reflex : Prop
```

We can also assume that `R` is reflexive:

```
Lego>[reflex:Reflex];
decl  reflex :Reflex
```

These terms may be dependent on the type `A` and relation `R`. Working in this context of a binary relation, we do not need to prefix by `[A:Type]`, `[R:A->A->Prop]`, `{A:Type}`, or `{R:A->A->Prop}` (say, in the case of `Reflex`); and we do not need to apply the declared identifiers (say `reflex`) or other constructions which otherwise have the abstraction forms. However, as `A` and relation `R` are *arbitrary*, we can use `Discharge` command to regain the polymorphism we hope to have originally.

```
Lego>Discharge A;
Discharge and forget..  A R reflex
```

The effect of successfully evaluating the command will forget all the variable declarations back to A (inclusive) but at the same time, 'parameterize' all the defined constants which are dependent on the deleted variables. For example, in the context after the above `Discharge A` command, the constant `Reflex` defined above becomes abstracted (polymorphic) with respect to A and R:

```
Reflex = [A:Type][R:A->A->Prop]{x:A}(R x x)
     : {A:Type}(A->A->Prop)->Prop
```

If there is another variable in the context on which `Reflex` is not dependent, it will be forgotten as far as `Reflex` is concerned.

To prevent a declaration or to allow a definition to be discharged in the future, one can use a dollar sign ($) immediately before the [of the declaration or definition. Here is an example.

```
[A|Type];
$[Eq:A->A->Prop];
$[B=Prop][C=Prop];
Discharge A;
```

After the above, A and B are forgotten while Eq and C still remain in the current context.

One may also indicate dependency of a declaration `$[M:A]` or a definition `[M=N]` over some declarations, say `[x1:A1]` and `[x2:A2]`, in the current context by `$[M:A//x1 x2]` or `[M=N//x1 x2]` where `x1` and `x2` are separated by a space. When `xi` (`i=1,2` in our example) is discharged, the declaration or definition is abstracted over `Ai`. For example, the parameterized inductive type `List` (see section 8) may be specifed as follows:

```
[A|Type(0)];
$[List : Type(0)//A];
$[nil  : List];
$[cons : A->List->List];
$[recL : {C:List->Type}
 {c:C nil}{f:{a:A}{l:List}(C l)->C (cons a l)}
 {l:List}C l];
```

```
[
[C:List->Type]
[c:C nil][f:{a:A}{l:List}(C l)->C (cons a l)]
[a:A][l:List]
   recL C c f nil ==> c
|| recL C c f (cons a l) ==> f a l (recL C c f l)
];
Discharge A;
```

7.3.6 Freeze and Unfreeze

`Freeze` and `Unfreeze` take one or more arguments which should occur in the current context as defined constants. They can be used either in a lego state or in a proof state. Evaluating `Freeze t1 ... tn` will stop the defined constants `ti` from being expanded, until an `Unfreeze ti` command is evaluated. `Unfreeze` can also have no arguments, in which case, it will unfreeze all previously frozen definitions.

7.4 Other commands

7.4.1 Help

In either a lego state or a proof state, you can use the following command

```
Help
```

which will print a brief list of basic commands and the basic syntax.

7.4.2 Logic

In a lego state, you may use command

```
Logic
```

to load the embedded higher-order logic for the type theory PCC or ECC (not for LF). The logical definitions obtained by evaluating `Logic` for ECC can be found in Appendix B.
Warning: use this command after you have chosen your type theory and in an empty context to avoid possible name clashes.

7.4.3 Include and ExportState

The commands `Include` and `ExportState` interact with Unix file system.

The command `Include` takes a string (viewed as a UNIX path to a file whose extension `.l` is optional) as its argument. The string may be any UNIX path not starting with '~'. It can be evaluated either in lego state or in proof state.

Evaluating `Include "path"` has the same effect as evaluating the text in the file specified by the path. (See Appendix C for more details on the UNIX environment variable `LEGOPATH`.) When the UNIX path is a single file name (*i.e.*, referring to a file in the current directory), the string delimiters '`"`' can be omitted. For example, if file `semigroup` is in the current directory (and `LEGOPATH` is set as its default),

```
Lego> Include "semigroup";
(* opening file ./semigroup.l *)
defn  Sig_SG = <A:Type(0)>A->A->A
      Sig_SG : Type(1)
... ... ...
(* closing file ./semigroup.l *)
Lego>
```

The command `ExportState` takes the same kind of arguments as `Include` and can be evaluated either in lego state or in proof state. Evaluating `ExportState filename` (or `ExportState "filename"`, or `ExportState "filename.l"`) will write the whole current proof state to the target file. This target file may be executed (in a shell) to recover to the system state when the `ExportState` command was executed.

7.4.4 Pwd and Cd

In a lego state or a proof state, the command `Pwd` prints the current working directory. The working directory can be changed with the command `Cd`.

7.4.5 echo, line, StartTimer and PrintTimer

`echo` and `line` are commands used to print comments and format proof documents. `StartTimer` and `PrintTimer` can be used to time the commands that come between them.

	Command	Meaning	Condition
`and`-introduction	`andI;`	`Refine pair;`	
`and`-elimination	`andE p;`	`Refine p; Intros _ _;`	
`or`-introduction	`orIL;`	`Refine inl;`	
	`orIR;`	`Refine inr;`	
`or`-elimination	`orE p;`	`Refine p;`	
`->`-introduction	`impI;`	`Intros _;`	
`->`-elimination	`impE p;`	`Refine p;`	`p : {x:A}B`
`not`-introduction	`notI;`	`Intros _;`	
`not`-elimination	`notE p;`	`Refine p;`	`p : {x:{y:A}B}C`
∀-introduction	`allI;`	`Intros _;`	
∀-elimination	`allE p;`	`Refine p;`	`p : {x:A}B`
∃-introduction	`exI P;`	`Refine ExIntro P;`	
∃-elimination	`exE p`	`Refine p;`	`p : Ex P`

Figure 1: Logical macros.

7.4.6 Macros for logical operators

There are several macro commands which can be used in a proof state for introduction and elimination of the usual logical operators as described in section 9.1 and Appendix B. They are not essential, but merely a convenient shorthand. These macros can only be used after the command `Logic` has been successfully evaluated (or the corresponding logical operators and its introduction and elimination operators have been defined as described in Appendix B).

The logical macros for introduction have the form `opI` (except that, for disjunction, there are two: `orIL` and `orE`) and those for elimination have the form `opE`. Figure 1 gives a description of each macro and its meaning (*i.e.*, it is essentially equivalent to evaluate the sequence of commands in the column of Meaning[2]) under the condition that the argument `p` has a type convertible to the form shown in the column of Condition.

Here is a simple example of the use of `andI`:

[2]Warning: The meanings given here are not exact — one should not literally replace one by the other.

```
  ?1 : and A B
Lego> andI;      (* use and-introduction *)
and Intro
  ... ...
  a : A
  b : B
  ?4 : A
  ?5 : B
Lego>
```

and a use of `andE`:

```
  ... ...
  p : and A B
  ?1 : A
Lego> andE p;     (* use and-elimination *)
and Elim
  ... ...
  p : and A B
  H : A
  H1 : B
  ?4 : A
Lego>
```

Note that the argument `P` for the macro `exI` is a propositional function which has to be supplied (in many cases, `exI ?;` will do). In fact, in many cases, it would be easier to use the commands given in the column of Meaning.

8 Inductive Data Types

A new feature of the Lego system, which is still under further development, is to allow the user to specify inductive data types like the type of natural numbers, the type of lists, *etc.*. Since the syntax for this mechanism is still subject to further changes, we only explain the general principle of specifying inductive data types and give some examples.

8.1 Specifying inductive data types

In general, a specification of a new (possibly parameterized) inductive data type consists of four parts:

1. Type formation: declaring the type (or type constructor) to be specified. (In the current implementation, this can in general be an arbitrary declaration.)

2. Constructor introduction: declaring the constructors which construct the (canonical) objects of the specified inductive type. (In the current implementation, this can in general be a list of arbitrary declarations.)

3. Eliminator introduction: declaring the elimination operator which usually plays the role of the recursion operator for the specified inductive type and gives the induction principle over the type.

   There is a general pattern to introduce this recursion (and induction) operator. Suppose that the inductive type to be specified is `A`. Then, its elimination operator, when applied to any family of types `C` of type `A->Type` and any objects which provide proofs (objects) of the types `C c` for the canonical objects `c` of type `A`, will return an object (proof) of type `{z:A}C z`.

4. Computation rules: specifying the reduction relation which gives the computation rules for the elimination operator when applied to the canonical objects of the type. The syntax for introducing computation (reduction) rules is:

   ```
   [ [x1:A1]...[xn:An]
     L1 ==> R1  ||  L2 ==> R2  ||  ...  ||  Lm ==> Rm
   ];
   ```

   where `[x1:A1]...[xn:An]` is a list of local declarations and `Li` and `Ri` are *patterns* in the current context (extended by the local context `[x1:A1]...[xn:An]`)[3] such that the variables declared in the local context (*i.e.*, `xi`) occurring in `Ri` also occur in `Li`. A defined variable occurring in `Li` is regarded as being in its expanded form, while a defined variable occurring in `Ri` is regarded as not expanded.

[3]A pattern in a context is either a variable in the context, or is a term of the form `P P1 ... Pn` such that `P` and `Pi` are patterns.

Inductive types in type theory have been studied in the literature by many people, including [CPM90][Dyb91][Luo92]. Warning: The mechanism for specifying new data types (especially new computation rules) in the current implementation is very liberal, and does not exclude semantically incorrect specifications.

8.2 Examples of inductive types

The following are some simple examples of specifying inductive types. For more examples and proof examples, see section 9.2.

8.2.1 the type of natural numbers

```
(* type formation *)
[N : Type(0)];
(* constructor introduction, canonical objects: zero and (succ n) *)
[zero : N];
[succ : N->N];
(* elimination operator *)
[recN : {C:N->Type}{a:C zero}{f:{x:N}(C x)->C (succ x)} {z:N}C z];
(* computation rules *)
[
[C:N->Type][a:C zero][f:{x:N}(C x)->C (succ x)][n:N]
   recN C a f zero ==> a
|| recN C a f (succ n) ==> f n (recN C a f n)
];
```

As a simple example of defining functions, we define the predecessor function `pred` and the addition function `plus` over natural numbers. (For more examples, see section 9.2.

```
Lego> [pred [n:N] = recN ([_:N]N) zero ([x,_:N]x) n  : N];
defn  pred = [n:N]recN ([_:N]N) zero ([x,_:N]x) n
      pred : N->N
Lego> [plus [m,n:N] = recN ([_:N]N) m ([_,x:N]succ x) n : N];
defn  plus = [m,n:N]recN ([_:N]N) m ([_,x:N]succ x) n
      plus : N->N->N
```

Now, we can verify that, for example, for any natural number `n`, `pred (succ n)` and `n` are convertible.

The following simple example (the first steps to prove the associativity of `plus`) shows how the induction principle given by the recursion operator can be used in a refinement proof:

```
Lego> Goal {m,n,l:N}Q (plus m (plus n l)) (plus (plus m n) l);
Goal
  ?0 : {m,n,l:N}Q (plus m (plus n l)) (plus (plus m n) l)
Lego> Intros m n l;
Intros (3) m n l
  m : N
  n : N
  l : N
  ?1 : Q (plus m (plus n l)) (plus (plus m n) l)
Lego> Refine recN [l:N] Q (plus m (plus n l)) (plus (plus m n) l);
Refine by  recN ([l:N]Q (plus m (plus n l)) (plus (plus m n) l))
  ?3 : Q (plus m (plus n zero)) (plus (plus m n) zero)
  ?4 : {x:N}(Q (plus m (plus n x)) (plus (plus m n) x))->
            Q (plus m (plus n (succ x))) (plus (plus m n) (succ x))
Lego>
```

The last step above uses `recN` to do induction on `l`, resulting in two new goals (base case and induction step) to solve. See section 9.2 for more examples.

8.2.2 finite types

Examples of specifications of finite types are given below.

```
(* empty type *)
[Empty : Type(0)];
[recE  : {C:Empty->Type}{z:Empty}(C z)];

(* unit type *)
[Unit : Type(0)];
[star : Unit];
[recU : {C:Unit->Type}(C star)->{z:Unit}(C z)];
[
[C:Unit->Type][c:C star]
recU C c star ==> c
];

(* type of booleans *)
[Bool : Type(0)];
[tt   : Bool];
[ff   : Bool];
[recB : {C:Bool->Type}(C tt)->(C ff)->{z:Bool}C z];

[
[C:Bool->Type][d:C tt][e:C ff]
      recB C d e tt ==> d
   || recB C d e ff ==> e
];
```

8.2.3 the type of lists

The inductive types in the following subsections can be specified for any universe `Type(i)`. For example, we may specify `Listi : Type(i)->Type(i)`. Here, we only give these examples for `Type(0)`. (Note: Do *not* use `Type` liberally in such cases; this could introduce logical paradoxes into the theory.)

```
[List : Type(0)->Type(0)];
[nil  : {A:Type(0)}List A];
[cons : {A|Type(0)}A->(List A)->(List A)];
[recL : {A|Type(0)}{C:(List A)->Type}
{c:C (nil A)}{f:{a:A}{l:List A}(C l)->C (cons a l)}
{l:List A}C l
];

[
[A:Type(0)][C:(List A)->Type]
[c:C (nil A)][f:{a:A}{l:List A}(C l)->C (cons a l)]
[a:A][l:List A]
   recL C c f (nil A) ==> c
|| recL C c f (cons a l) ==> f a l (recL C c f l)
];
```

8.2.4 disjoint sum

```
[Sum  : Type(0)->Type(0)->Type(0)];
[i    : {A:Type(0)}{B:Type(0)}A->(Sum A B)];
[j    : {A:Type(0)}{B:Type(0)}B->(Sum A B)];
[case : {A:Type(0)}{B:Type(0)}{C:(Sum A B)->Type}
({x:A}(C (i A B x)))->({y:B}(C (j A B y))) ->
{z:(Sum A B)}(C z)
];
[
[A:Type(0)][B:Type(0)][C:(Sum A B)->Type]
[f:{x:A}(C (i A B x))][g:{y:B}(C (j A B y))][a:A][b:B]
case A B C f g (i A B a) ==> f a  ||
case A B C f g (j A B b) ==> g b
];
```

8.2.5 well-ordering types

```
[W   : {A:Type(0)}(A->Type(0))->Type(0)];
[sup : {A:Type(0)}{B:A->Type(0)}{x:A}((B x)->(W A B))->(W A B)];
[recW: {A:Type(0)}{B:A->Type(0)}{C:(W A B)->Type}
       ({x:A}{y:(B x)->(W A B)}({v:(B x)}(C (y v)))->(C (sup A B x y))) ->
       {z:(W A B)}(C z)
];
(* kappa is an auxiliary function *)
[kappa   [A:Type(0)][B:A->Type(0)][C:(W A B)->Type]
 [f:{x:A}{y:(B x)->(W A B)}({v:(B x)}(C (y v)))->(C (sup A B x y))]
 [x:A][y:(B x)->(W A B)]
       = [v:(B x)](recW A B C f (y v))
];
[
[A:Type(0)][B:A->Type(0)][C:(W A B)->Type][a:A][b:(B a)->(W A B)]
[f:{x:A}{y:(B x)->(W A B)}({v:(B x)}(C (y v)))->(C (sup A B x y))]
recW A B C f (sup A B a b) ==> f a b (kappa A B C f a b)
];
```

9 Examples

In this section, some examples of problem formalization, proof development and program specification and development are given. Some of them (*e.g.*, examples in the basic logic in section 9.1 and basic examples for inductive types in section 9.2) are simple examples together with some exercises to help the beginner to start playing with the system. Most of the examples in section 9.2 are given by Pollack. The mechanism for program specification and stepwise development of programs by refinement is described in section 9.3, which is based on Burstall's idea on deliverables [BM91][McK92] and the work described in [Luo91b]. For examples of LF style of formalization of logics, we refer to [AHMP92]. The example in section 9.4, given by Pollack, shows another way of formalizing logics. A proof of the Schroëder-Bernstein theorem, given by Hofmann, can be found in section 9.5.

9.1 Logical examples — the embedded logic

By the Curry-Howard principle of formulas-as-types, there is an embedded (intuitionistic) higher-order logic in the (various versions of) calculus of constructions. Putting in another way, we can define (interpret) the logical constructs in the type theory in an appropriate way so that, for example, the usual inference rules are all admissible.[4] In such an interpretation, logical formulas are interpreted as propositions (types of type `Prop`). The provability is defined as follows. A proposition `P` is *provable* (in the context `C`) if, and only if, `P` is *inhabited* (in `C`), that is, there exists a term `M` such that `M` is of type `P` (in `C`).

In this section, we give examples to show how logical connectives and quantifiers can be defined and how the usual logical inference rules are proved to be valid. These examples are fairly simple and can also be used as exercises for beginners to start playing with the system.

9.1.1 logical connectives

The usual logical connectives and constants can be defined as follows.

1. *Implication*: Intuitionistic implication is interpreted just as the arrow `P->Q` when both `P` and `Q` are propositions. The usual implication introduction rule is

[4]This embedded logic is a conservative extension of the intuitionistic higher-order logic (*c.f.*, [Chu40][Tak75]). See [Luo91c].

then reflected by the λ-introduction rule in the type system. We can show, for example, Modus Ponens as follows:

```
(* Modus Ponens *)
 Goal {P,Q:Prop}(P->Q)->P->Q;
Intros P Q h; Immed;
```

The developed proof is `[P,Q:Prop][h:P->Q]h`.

The reader might like to use the usual logical axioms like `{P,Q:Prop}P->Q->P`, `{P,Q,R:Prop}(P->Q)->(Q->R)->P->R` as exercises in proof development.

2. *Conjunction*: The logical conjunction operator can be defined by coding the elimination rule in the following way:

```
(* and-operator *)
 and [P,Q:Prop] = {R:Prop}(P->Q->R)->R ;
```

Then, the ordinary and-introduction/elimination rules can be proved as follows:

```
(* and-introduction -- and_intro *)
 Goal {P,Q:Prop}P->Q->(and P Q);
Intros P Q p q; Expand and; Intros R h; Refine h; Immed;
Save and_intro;
(* and-elimination (right) -- and_elim_right *)
 Goal {P,Q:Prop}(and P Q)->Q;
Intros P Q; Expand and; Intros h; Refine h; Intros p q; Refine q;
Save and_elim_right;
```

3. *Disjunction*: The logical disjunction operator can be defined as

```
(* or-operator *)
 or [P,Q:Prop] = {R:Prop}(P->R)->(Q->R)->R ;
```

The introduction/elimination rules can be proved as follows:

```
(* or-introduction (left) -- or_intro_left *)
 Goal {P,Q:Prop}P->(or P Q);
Intros P Q p; Expand or; Intros R f g; Refine f p;
Save or_intro_left;
(* or-elimination -- or_elim *)
 Goal {P,Q:Prop}(or P Q)->{R:Prop}(P->R)->(Q->R)->R;
Intros P Q orPQ; Refine orPQ; Save or_elim;
```

The reader may try to prove from these as exercises the commutativity law `{P,Q:Prop}(or P Q)->(or Q P)` and the associativity laws (*e.g.*, `{P,Q,R:Prop}(or (or P Q) R)->(or P (or Q R))`).

4. *Falsity and negation*: The logical constant FALSE is interpreted as the 'empty' proposition `{P:Prop}P`. (It is a meta theorem about the type theory that `{P:Prop}P` is not inhabited.) Negation can be interpreted by intuitionistic refutation.

```
(* falsity and negation -- false and not *)
 false = {P:Prop}P;  not [P:Prop] = P->false ;
```

It is very easy to show, for example, that `false` implies every proposition, that is `{P:Prop}false->P` is inhabited.

5. *Truth*: The logical constant TRUE can be interpreted as any (fixed) closed proposition inhabited in the empty context, *e.g.*, `{P:Prop}P->P`.

9.1.2 quantifiers

The universal quantifier is interpreted as the Π-operator in the type theory. That is, $\forall x \in A.P$ is interpreted as `{x:A}P`, where `P` is a proposition under the assumption `x` is of type `A`. The introduction rule for universal quantifier is reflected in the λ-introduction rule and the elimination rule by the application rule in the type theory, respectively.

The existential quantifier ($\exists x \in A.P(x)$) can be defined in the following way (usually called *weak sum*):

```
(* existential quantifier *)
 exists [A:Type][P:A-Prop] = {R:Prop}({x:A}(P x)->R)->R : Prop ;
```

Existential introduction/elimination can be proved as follows:

```
(* existential introduction -- exists_intro *)
 Goal {A:Type}{P:A->Prop}{x:A}(P x)->(exists A P);
Intros A P a pa; Expand exists; Intros R h; Refine h; Refine a; Refine pa;
Save exists_intro;
(* existential elimination -- exists_elim *)
 Goal {A:Type}{P:A->Prop}(exists A P)->{R:Prop}({x:A}(P x)->R)->R;
Intros A P exAP; Refine exAP; Save exists_elim;
```

9.1.3 Leibniz's equality

We show how Leibniz's equality can be defined and prove that it is an equivalence relation.

```
(* Leibniz's equality *)
(* Argument synthesis is used *)
 Q [A|Type][x,y:A] = {P:A->Prop}(P x)->(P y) ;
(* Reflexivity -- Q_reflex *)
 Goal {A:Type}{x:A}Q x x;
Intros A a; Expand Q; Intros P pa; Immed;
Save Q_reflex;
(* Symmetry -- Q_sym *)
(* trick used in the proof *)
 Goal {A:Type}{x,y:A}(Q x y)->(Q y x);
Intros A a b qab; Expand Q; Intros P pb; Refine qab [x:A](P x)->(P a);
Intros pa; Refine pa; Immed;
Save Q_sym;
(* Transitivity -- Q_trans *)
(* proved by function composition *)
 Goal {A:Type}{x,y,z:A}(Q x y)->(Q y z)->(Q x z);
Intros A a b c qab qbc;
Expand Q; Intros P pa; Refine qbc; Refine qab; Immed;
Save Q_trans;
```

9.2 Some definitions and proofs about inductive data types

We define some functions on some basic inductive data types (see section 8) and prove some of their properties. We also leave some as exercises. In the following, we assume that `XCC` is initiated, command `Logic` is executed, and the data types in section 8 have been specified.

9.2.1 functions and properties for natural numbers

```
(* functions and relations on natural numbers *)
[pred [n:N] = recN ([_:N]N) zero ([x,_:N]x) n  : N];
[plus [m,n:N] = recN ([_:N]N) m ([_,x:N]succ x) n : N];
[minus[m,n:N] = recN ([_:N]N) m ([_,x:N]pred x) n : N];
```

```
[subtract [m,n:N] = recN ([_:N]N) m ([_,x:N]pred x) n];
[le   [m,n:N] = Ex [k:N] (Q (plus m k) n) : Prop ];
[less [m,n:N] = and (le m n) (not (Q m n)) : Prop ];

(* Exercise: Define the multiplication function over natural numbers. *)

(* properties of natural numbers *)

(* n is either zero or a successor *)
Goal {n:N}or (Q n zero) (Ex [k:N] (Q n (succ k)));
Intros n;
[C [n:N] = or (Q n zero) (Ex [k:N] (Q n (succ k))) : Prop ];
Refine recN C; Refine inl; Refine Q_refl;
Intros x cx; Refine inr; Refine ExIntro; Refine x; Refine Q_refl;
Save zero_or_succ;

(* Peano's fourth axiom *)
Goal {n:N}not (Q zero (succ n));
Intros n h;
[C = recN ([_:N]Prop) ({x:Prop}x->x) [n:N][P:Prop]absurd ];
Claim Q ({x:Prop}x->x) absurd; Refine ?+1 [P:Prop]P; Refine [P:Prop][x:P]x;
Equiv Q (C zero) (C (succ n)); Refine Q_resp; Refine h;
Save peano4;

(* Exercises: for the second, you may use peano4 and peano3 *)
Goal {m,n,l:N}Q (plus m (plus n l)) (plus (plus m n) l);  ... Save peano3;
Goal {n:N}not (Q n (succ n));  ... Save notSucc;

(* plus is associative *)
Goal {m,n,l:N}Q (plus m (plus n l)) (plus (plus m n) l);
Intros m n l;
[C [l:N] = Q (plus m (plus n l)) (plus (plus m n) l) : Prop];
Refine recN C; Refine Q_refl;
Intros x cx;
Expand C; Equiv Q (succ (plus m (plus n x))) (succ (plus (plus m n) x));
Refine Q_resp; Refine cx;
```

```
Save plusAss;

(* Exercises *)
Goal {m,n:N}Q (plus (succ m) n) (plus m (succ n)); ... Save lemmaPlus;
(* plus is commutative *)
Goal {m,n:N}Q (plus m n) (plus n m); ... Save plusComm;
(* zero is the unit for plus *)
Goal {x,k:N}(Q (plus x k) x)->(Q k zero); ... Save zeroUnit;
Goal {m,n:N}(Q (plus m n) zero)->(Q n zero); ... Save plusStrict;
(* some properties about subtract *)
Goal {x,y:N}Q (subtract x (succ y)) (subtract (pred x) y);
Goal {x,y:N}Q (subtract (succ x) (succ y)) (subtract x y);
Goal {x:N}Q (subtract x x) zero;
Goal {x:N}Q (subtract zero x) zero;


(* some properties of le and less *)
Goal {m:N}le (pred m) m;
Intros m; Refine recN [m:N]le (pred m) m;
Refine ExIntro; Refine zero; Refine Q_refl;
Intros x p; Equiv le x (succ x);
Refine ExIntro; Refine succ zero; Refine Q_refl;
Save lePred;
(* antisymmetry of le *)
Goal {m,n:N}(le m n)->(le n m)->(Q m n);
Intros m n p q; Refine p; Refine q; Intros k1 ek1 k2 ek2; Refine ek2;
Claim Q k2 zero; Refine (Q_sym ?+1) [k2:N]Q m (plus m k2); Refine Q_refl;
Refine plusStrict; Refine k1; Refine zeroUnit n (plus k1 k2);
Refine ek2; Refine Q_trans; Refine plus (plus n k1) k2;
Refine (plusAss n k1 k2); Refine Q_resp [z:N]plus z k2; Refine ek1;
Save leAntiSym
(* transitivity of le *)
Goal {m,n,l:N}(le m n)->(le n l)->(le m l);
Intros m n l p q; Refine q; Refine p;
Intros i p1 j q1; Refine ExIntro; Refine plus i j;
Refine q1; Refine Q_trans; Refine (plus (plus m i) j);
Refine plusAss; Refine Q_resp ([x:N]plus x j); Refine p1;
Save leTrans;

(* Exercises: *)
Goal {m,n:N}(le m n)->(not (Q m (succ n)));
Goal {P:N->Prop}{m:N}
     (and (P m) ({i:N}(le i (pred m))->(P i))) -> {i:N}(le i m)->(P i);
Goal {m,n:N}iff (less m n) (Ex [k:N]and (Q (plus m k) n) (not (Q k zero)));
Goal {n:N}not (less n zero);  ... Save less0;
Goal {m,n:N}(less n (pred m))->(less n m);  ... Save less1;
Goal {P:N->Prop}{m:N}
     (and (P (pred m)) ({i:N}(less i (pred m))->(P i))) ->
     {i:N}(less i m)->(P i);


(* double induction principle for natural numbers *)
Goal {C:N->N->Type}
     (C zero zero) ->
     ({y:N}(C zero y)->(C zero (succ y))) ->
     ({x:N}({y:N}C x y)->C (succ x) zero) ->
     ({x:N}({y:N}C x y)->{y:N}(C (succ x) y)->C (succ x) (succ y)) ->
     {x,y:N}C x y;
intros C bb_step bs_step sb_step ss_step;
Refine recN [x:N]{y:N}C x y; Refine recN (C zero); Immed;
intros x h; Refine recN (C (succ x));
Refine sb_step x h; Refine ss_step x h;
Save doubleInduct;
```

9.2.2 functions and properties for booleans

```
(* some functions over booleans *)
[is_tt [b:Bool] = Q b tt];
[is_ff [b:Bool] = Q b ff];
[if [a:Bool][D|Type][d,e:D] = recB ([_:Bool]D) d e a];
[andd [a,b:Bool] = if a b ff];
[orr  [a,b:Bool] = if a a b];
[nott [a:Bool] = if a ff tt];
```

```
[impp [a,b:Bool] = if b tt (if a ff tt)];

(* double induction principle for booleans *)
Goal {C:Bool->Bool->Type}
     (C tt tt)->(C tt ff)->(C ff tt)->(C ff ff)->{x,y:Bool}C x y;
intros _____; Refine recB [x:Bool]{y:Bool}C x y;
Refine recB [y:Bool]C tt y; Immed;
Refine recB [y:Bool]C ff y; Immed;
Save Bool_double_induct;
(* tt is not equal to ff *)
Goal not (Q tt ff);
Intros p; [C = recB ([_:Bool]Prop) ({x:Prop}x->x) absurd];
Claim Q ({x:Prop}x->x) absurd; Refine ?+1 [P:Prop]P; Refine [P:Prop][x:P]x;
Equiv Q (C tt) (C ff); Refine Q_resp; Refine p;
Save tt_not_ff;

(* Exercises: *)
Goal {b:Bool}or (is_tt b) (is_ff b); ... Save tt_or_ff;
Goal {A|Type}{a,b,c|A}{x|Bool}(Q (if x a b) c)->or (Q a c) (Q b c);
Goal {A|Type}{a,b,a',b'|A}
     (Q a a')->(Q b b')->{x:Bool}Q (if x a b) (if x a' b');
Goal {a:Bool}Q (andd a a) a;
Goal {b:Bool}iff (is_tt (nott b)) (not (is_tt b));
Goal {a,b|Bool}((is_tt a)->(is_tt b))->(is_ff b)->(is_ff a);
Goal {a,b:Bool}iff (is_tt (orr a b)) (or (is_tt a) (is_tt b));

(* associativity of orr *)
Goal {a,b,c:Bool}Q (orr (orr a b) c) (orr a (orr b c));
intros; Refine tt_or_ff a;
intros tta; Qrepl tta; Refine Q_refl;
intros ffa; Qrepl ffa; Refine Q_refl;
Save orr_associates;
(* commutativity of orr *)
Goal {a,b:Bool}Q (orr a b) (orr b a);
intros; Refine tt_or_ff a;
intros ah; Qrepl ah; Refine tt_or_ff b;
intros bh; Qrepl bh; Refine Q_refl;
intros bh; Qrepl bh; Refine Q_refl;
intros ah; Qrepl ah; Refine tt_or_ff b;
intros bh; Qrepl bh; Refine Q_refl;
intros bh; Qrepl bh; Refine Q_refl;
Save orr_commutes;

(* deMorgan laws *)
Goal {a,b:Bool}iff (is_ff (orr a b)) (and (is_ff a) (is_ff b));
intros; Refine pair; intros hyp; Refine pair;
Refine tt_or_ff a; intros tta; Refine tt_not_ff;
Qrepl Q_sym hyp; Qrepl tta; Refine Q_refl;
intros; Immed; Refine tt_or_ff b; intros ttb; Refine tt_not_ff;
Qrepl Q_sym hyp; Qrepl orr_commutes a b; Qrepl ttb; Refine Q_refl;
intros; Immed; intros g; Qrepl fst g; Qrepl snd g; Refine Q_refl;
Save deMorgan;

(* proof by contradiction *)
Goal {a|Bool}(is_tt a)->(is_ff a)->{A:Prop}A;
intros; Refine tt_not_ff; Qrepl Q_sym H; Qrepl Q_sym H1; Refine Q_refl;
```

9.2.3 functions and properties for lists

```
(* functions about lists *)
[hd [A|Type(0)][a:A][l:List A] =
    recL ([_:List A]A) a ([x:A][_:List A][_:A]x) l];
[tl [A|Type(0)][l:List A] =
    recL ([_:List A]List A) (nil A) ([_:A][k,_:List A]k) l];
[append [A|Type(0)][k,l:List A] =
    recL ([_:List A]List A) l ([a:A][_,j:List A]cons a j) k];
[unit [A|Type(0)][a:A] = cons a (nil A)];
[length [A|Type(0)][l:List A] =
    recL ([_:List A]N) zero ([_:A][_:List A]succ) l];
[exist [A|Type(0)][P:A->Bool] : (List A)->Bool =
    recL ([_:List A]Bool) ff ([b:A][_:List A][rest:Bool]orr (P b) rest)];
[member [A|Type(0)][eq:A->A->Bool][a:A] : (List A)->Bool = exist (eq a)];
[nth [n:N][A|Type(0)][l:List A][a:A] : A =
     [zro [k:List A] = hd a k]
     [suc [_:N][f:(List A)->A][k:List A] = f (tl k)]
       recN ([_:N](List A)->A) zro suc n l];
[map [A|Type(0)][f:A->A] : (List A)->(List A) =
     recL ([_:List A]List A) (nil A) ([b:A][_:List A][k:List A]cons (f b) k)];

(* double induction for lists *)
Goal {A|Type(0)}{C:(List A)->(List A)->Type}
     (C (nil A) (nil A))->
     ({b:A}{l:List A}(C (nil A) l)->C (nil A) (cons b l))->
     ({b:A}{x:List A}({y:List A}C x y)->C (cons b x) (nil A))->
     ({b:A}{x:List A}({y:List A}C x y)->{c:A}{y:List A}(C (cons b x) y)->
                    C (cons b x) (cons c y))->
     {x,y:List A}C x y;
intros A C Bool bs sb ss; Refine recL [x:List A]{y:List A}C x y;
Refine recL (C (nil A)); Immed;
intros b x h; Refine recL (C (cons b x));
Refine sb b x h; Refine ss b x h;
Save list_double_induct;

(* Exercises: *)
Goal {A|Type(0)}{l:List A}{a:A}not (Q (nil A) (cons a l));
... Save nil_not_cons;
Goal {A|Type(0)}{l:List A}{a:A}not (Q l (cons a l));
... Save l_not_consl;


(* associativity of append *)
Goal {A|Type(0)}{j,k,l:List A}
     Q (append j (append k l)) (append (append j k) l);
intros _;
Refine recL [j:List A]{k,l:List A}
       Q (append j (append k l)) (append (append j k) l);
intros; Refine Q_refl;
intros;
Equiv Q (cons a (append l (append k l1)))
```

```
        (cons a (append (append l k) l1));
Qrepl H k l1; Refine Q_refl;
Save append_associative;

(* Exercises: *)
Goal {A|Type(0)}{P:A->Bool}{j,k:List A}
     Q (exist P (append j k)) (orr (exist P  j) (exist P k));
Goal {A|Type(0)}{eq:A->A->Bool}{a:A}{j,k:List A}
     Q (member eq a (append j k)) (orr (member eq a j) (member eq a k));
```

9.3 Program specification and data refinement

In this section, the specification mechanism based on the notion of deliverables are implemented in Lego. (See [BM91][Luo91b].) A notion of mathematical theory for structured abstract reasoning can also be implemented similarly (see [Luo91a][LPT89]).

In the following, we assume that (1) ECC has been initiated; (2) `Logic` has been included (see section 7.4.2 and Appendix B); (3) the inductive type of natural numbers, `N`, has been specified (see section 8) and the examples and exercises in section 9.2.1 have been done.

First, some preliminary definitions.

```
[and5 [A,B,C,D,E:Prop] = {X:Prop}(A->B->C->D->E->X)->X : Prop];
[pair5 = [A,B,C,D,E:Prop][a:A][b:B][c:C][d:D][e:E]
         [X:Prop][h:A->B->C->D->E->X]h a b c d e: and5 A B C D E];
(* Binary relations, equivalence relations *)
[Rel [A:Type] = A->A->Prop];
[Equivalence [A|Type][R:Rel A] = and3 R.refl R.sym R.trans];
[EQUIVALENCE [A:Type] = <R:Rel A>Equivalence R];
[rel [A|Type][R:EQUIVALENCE A] = R.1 : Rel A];
(* setoids --- pairs of a type and a binary relation over the type *)
[SETOID = <A:Type>Rel A]; [dom [S:SETOID] = S.1]; [eq [S|SETOID] = S.2];
```

A *specification* in the type theory consists of (a pair of) a type, whose objects are the possible structures (program modules) which may realize the specification, and a predicate over the structure type, which specifies the properties that any realization should satisfy. A *realization* of a specification is a structure (an object of the structure type) which satisfies the required properties.

```
(* general structure of specifications *)
[STR = Type];                      (* class of structure types *)
[Spec [S:STR] = S->Prop];          (* class of S-specifications *)
[SPEC = <S:STR>Spec S];            (* class of specifications *)
[SP:SPEC];
[Str = SP.1];                      (* structure-type for SP *)
[Ax  = SP.2];                      (* axioms for SP *)
[Mod = <a:SP.1>(SP.2 a) : Type]; (* realizations of SP with proofs *)
Discharge SP;
```

A specification SP' is an *implementation* of another SP through a refinement map `refn` (a function from the structure type of SP' to that of SP) if the images of `refn` over the realizations of SP' are realizations of SP.

```
(* type-theoretic notion of implementation *)
[Sat [SP,SP'|SPEC][f:SP'.Str->SP.Str]
   = {s':SP'.Str}(SP'.Ax s')->(SP.Ax (f s'))];
[IMPL [SP,SP':SPEC] = <f:SP'.Str->SP.Str> (Sat f) ];
[SP,SP'|SPEC]; [impl:IMPL SP SP'];
[refn = impl.1];                    (* refinement map *)
[sat = impl.2];                     (* proof of satisfaction condition *)
Discharge SP;
```

Examples of specifications for abstract data types are specifications of stacks and arrays of natrual numbers.

```
(* the specification of stacks *)
(* the structure type *)
[Str_STACK = <Stack:SETOID><empty:Stack.dom><push:N->Stack.dom->Stack.dom>
             <pop:Stack.dom->Stack.dom>Stack.dom->N];
[SS|Str_STACK]; [Stack = SS.1];
[empty = SS.2.1]; [push = SS.2.2.1]; [pop = SS.2.2.2.1]; [top = SS.2.2.2.2];
(* axioms *)
[RespSTACKpush = {n,m:N}{s,s':Stack.dom}
                 (Q n m)->(eq s s')->eq (push n s) (push m s')];
[RespSTACKpop = {s,s':Stack.dom}(eq s s')->(eq (pop s) (pop s'))];
[RespSTACKtop = {s,s':Stack.dom}(eq s s')->(Q (top s) (top s'))];
[RespSTACK = and3 RespSTACKpush RespSTACKpop RespSTACKtop];
```

```
[EqSTACK = and (Equivalence (eq|Stack)) RespSTACK];
[AxSTACK1 = eq (pop empty) empty];
[AxSTACK2 = Q (top empty) zero];
[AxSTACK3 = {n:N}{s:Stack.dom}eq (pop (push n s)) s];
[AxSTACK4 = {n:N}{s:Stack.dom}Q (top (push n s)) n];
[Ax_STACK = and5 EqSTACK AxSTACK1 AxSTACK2 AxSTACK3 AxSTACK4];
Discharge SS;
STACK = (Str_STACK,Ax_STACK) : SPEC];

(* specification of arrays *)
(* structure type *)
[Str_ARRAY = <Array:SETOID><newarray:Array.dom>
             <assign:N->Array.dom->N->Array.dom>Array.dom->N->N];
[A|Str_ARRAY]; [Array = A.1];
[newarray = A.2.1]; [assign = A.2.2.1]; [access = A.2.2.2 : Array.dom->N->N];
(* axioms *)
[RespARRAYassign = {n,m,i,j:N}{a,a':Array.dom}
     (Q n m)->(eq a a')->(Q i j)->(eq (assign n a i) (assign m a' j))];
[RespARRAYaccess = {a,a':Array.dom}{i,j:N}
     (eq a a')->(Q i j)->(Q (access a i) (access a' j))];
[RespARRAY = and RespARRAYassign RespARRAYaccess];
[EqARRAY = and (Equivalence (eq|Array)) RespARRAY];
[AxARRAY1 = {i:N}Q (access newarray i) zero];
[AxARRAY2 = {n,i,j:N}{arr:Array.dom}
     and ((Q i j)->(Q (access (assign n arr i) j) n))
         ((not (Q i j))->(Q (access (assign n arr i) j) (access arr j)))];
[Ax_ARRAY = and3 EqARRAY AxARRAY1 AxARRAY2];
Discharge A;
[ARRAY = (Str_ARRAY,Ax_ARRAY) : SPEC];
```

Now, we give a proof that `ARRAY` is an implementation of `STACK` through the following refinement map `refn_ARRAYtoSTACK`, by which a stack is represented by an array-pointer pair.

```
(* the refinement map, constructed by proof *)
Goal ARRAY.Str -> STACK.Str;
Intros A; Intros #;
```

```
(* domain of impl of stack *)
Intros #; [domImpl = (Array|A).dom#N]; Refine domImpl;
(* equality over the impl domain *)
Refine [s,s':domImpl]
   and (Q s.2 s'.2) ({i:N}(less i s.2)->(Q (access s.1 i) (access s'.1 i)));
Normal;
(* empty stack *)
Intros #; Refine (newarray|A,zero);
(* push *)
Intros #; Refine [n:N][arri:domImpl]((assign n arri.1 arri.2),(succ arri.2));
(* pop *)
Intros #; Refine [arri:domImpl](arri.1,(pred arri.2));
(* top *)
[Ifzero [A|Type][a,b:A] = recN ([_:N]A) a ([_:N][_:A]b)];
Refine [ai:domImpl] Ifzero zero (access ai.1 (pred ai.2)) ai.2;
(* QED *)
Save refn_ARRAYtoSTACK;
```

To prove the implementation correctness, we first prove the following lemmas, the proofs of most of which are left as exercises.

```
(* Lemmas *)
Goal {A:ARRAY.Str}{ax_arrayA : ARRAY.Ax A}
     Equivalence (eq|(Stack|(refn_ARRAYtoSTACK|A)));
... Save equivSTACK;  (* exercise *)

Goal {A|ARRAY.Str}{ax_arrayA : ARRAY.Ax A} AxSTACK1|(refn_ARRAYtoSTACK|A);
Intros A ax_arrayA; Expand AxSTACK1; Refine pair; Refine Q_refl;
Intros i lessipred0; Refine less0; Refine i; Refine less1; Refine lessipred0;
(* QED *)
Save axSTACK1;

(* for i=2,3,4, prove the following *)
Goal {A|ARRAY.Str}{ax_arrayA : ARRAY.Ax A} AxSTACKi|(refn_ARRAYtoSTACK|A);
... Save axSTACKi;  (* exercises *)
Goal {A|ARRAY.Str}{ax_arrayA : ARRAY.Ax A} RespSTACK|(refn_ARRAYtoSTACK|A);
... Save respSTACK; (* exercise *)
```

Now, the correctness proof.

```
(* correctness proof of the abstract implementation *)
Goal Sat refn_ARRAYtoSTACK;
Intros A ax_arrayA; Refine pair5;
(* EqSTACK *)
Refine pair; Refine equivSTACK A ax_arrayA;
(* RespSTACK *)
Refine respSTACK ax_arrayA;
(* AxSTACK1-4 *)
Refine axSTACK1 ax_arrayA;
Refine axSTACK2 ax_arrayA;
Refine axSTACK3 ax_arrayA;
Refine axSTACK4 ax_arrayA;
(* QED *)
Save sat_refn_ARRAYtoSTACK;
```

Various specification operations can be defined and used for structured specifcation and modular design by refinement.

```
(* specification operations *)
(* Cross *)
[Cross [SP,SP':SPEC] =
   (SP.Str#SP'.Str, [s:SP.Str#SP'.Str]and (SP.Ax s.1) (SP'.Ax s.2)) : SPEC];
(* Sigma *)
[Sigma [SP:SPEC][P:SP.Str->SPEC] =
   (<s:SP.Str>(P s).Str,
    [s:<s:SP.Str>(P s).Str]and (SP.Ax s.1) ((P s.1).Ax s.2)) : SPEC];
(* Pi *)
[Pi [SP:SPEC][P:SP.Str->SPEC] =
   ({s:SP.Str}(P s).Str,
    [f:{s:SP.Str}(P s).Str]{s:SP.Str}(SP.Ax s)->((P s).Ax (f s))) : SPEC];
(* Con, constructors --- image constructed *)
[Con [S,S'|STR][k:S'->S][P':S'.Spec] =
     [s:S] Ex [s':S'] and (P' s') (Q (k s') s) : S.Spec];
(* Sel, selectors --- inverse image selected *)
[Sel [S,S'|STR][j:S'->S][P:S.Spec] = [s':S'] P (j s') : S'.Spec];
(* Abs, abstractors --- observational absrtaction *)
[Abs [S|STR][R:EQUIVALENCE S][P:S.Spec] =
     [s:S] Ex [s':S] and (P s') (R.rel s s') : S.Spec];

(* Other operations definable using the above *)
(* Join *)
[Join [S|STR][SP,SP':S.Spec] =
   Sel ([s:S](s,s)) (Cross (S,SP:SPEC) (S,SP':SPEC)).Ax];
(* Extend *)
[Extend [SP:SPEC][ext_str:SP.Str->STR][S'=<s:SP.Str>s.ext_str][ext_ax:S'->Prop]
      = (S',Join (Sel ([s':S']s'.1) SP.Ax) ext_ax) : SPEC];
(* Hiding *)
[Hide [S|STR][SP:SPEC][f:SP.Str->S] = (S, Con f SP.Ax) : SPEC ];
```

Parameterized specifications and a notion of implementation between them can be defined as follows.

```
(* parameterized specs with arbitrary parameter type Par *)
[PAR_SPEC [Par:Type] = Par->SPEC];
[Sat_PAR [Par|Type][F,G:PAR_SPEC Par][d:{s:Par}(G s).Str->(F s).Str]
   = {s:Par} Sat (d s)];
(* parameterized specification over specifications *)
[ParSPEC [S,S':STR] = S.Spec -> S'.Spec];
[Impl_Par [S,S'|STR][F,G:ParSPEC S S'] = {P:S.Spec} Impl (F P) (G P)];
[Sat_Par  [S,S',S''|STR][F:ParSPEC S S'][G:ParSPEC S S''][f:S''->S'] =
   {P:S.Spec}Sat|(S',(F P):SPEC)|(S'',(G P):SPEC) f];
```

9.4 Deduction theorem for the minimal propositional logic

This section contains an example by Pollack of proving the deduction theorem for a formal presentation of the minimal logic in LEGO.

```
(** Deduction Theorem for Minimal Implicational Logic **)

(******************************************
Similar formalization has been carried out for a full presentation of
first-order logic.  In order to keep this example small and simple, we
restrict to Minimal Implicational Logic.  In order to avoid speaking
of "sets of assumptions", we define entailment from a set of exactly
two asumptions, and allow duplicate assumptions.
**********************************************)

Init PCC;
(** The object logic: Syntax **)
[o:Prop]        (* 'o' is the type of propositions *)
[arr:o->o->o];  (* 'arr' is the implication *)
(** The object logic: Axioms and Rule **)
(* define schemas for the axioms (i.e. the types of K and S) *)
[k [A,B:o] = arr A (arr B A)]
[s [A,B,C:o] = arr (arr A (arr B C))(arr (arr A B) (arr A C))]
(* Define the 'entailment' relation
 * Hypotheses G and H entail X if X has every property shared by
 * G, H, and every instance of K and S, and preserved by Modus Ponens
 *)
[entails [G,H,X:o] =
         {P:o->Prop}
         (P G)->                                  (* G has property P *)
         (P H)->                                  (* H has property P *)
         ({a,b:o}P (k a b))->                (* every instance of K has P *)
         ({a,b,c:o}P (s a b c))->            (* every instance of S has P *)
         ({a,b|o}(P (arr a b))->(P a)->(P b))->   (* MP preserves P *)
         (P X) ];                            (* ... then X has P *)
(** The Abstraction Algorithm **)
(* lam x.x = app (app S K) K *)
Goal {G,H,A:o}entails G H (arr A A);
Intros G H A P g h K S MP;
Refine MP (MP (S ? ? ?) (K ? ?)) (K ? ?);
Refine A;
Save lemma_identity;
(* lam x.m = app K m if x not free in m *)
Goal {G,H,A,M:o}(entails G H M)->(entails G H (arr A M));
Intros G H A M m P g h K S MP;
Refine MP (K ? ?); Refine m; Immed;
Save lemma_forget;
(* lam x.UV = app (app S (lam x.U)) (lam x.V) *)
Goal {G,H,A,M,N|o}(entails G H (arr A (arr M N)))->
```

```
                  (entails G H (arr A M))->(entails G H (arr A N));
Intros G H A M N h g P H1 H2 K S MP;
Refine MP; Refine 4 MP (S ? ? ?);Refine 11 h; Immed; Refine 5 g; Immed;
Save lemma_apply;

   (*** The Deduction Theorem ***)

Goal {G,A,M:o}(entails G A M)->(entails G G (arr A M));
Intros G A M hyp;
(* given a derivation of 'G,A entail M', transform every node
 * to (arr A node)
 *)
Refine hyp [node:o]entails G G (arr A node);
(* gives 5 cases .. *)
(* case 1: node is G *)
Refine lemma_forget; Intros P H1 H2 K S MP; Refine H1;
(* case 2: node is A *)
Refine lemma_identity;
(* case 3: node is K *)
Intros a b; Refine lemma_forget; Intros P H1 H2 K S MP; Refine K;
(* case 4: node is S *)
Intros a b c; Refine lemma_forget; Intros P H1 H2 K S MP; Refine S;
(* case 5: node follows from previous nodes by MP *)
Refine lemma_apply;
Save dedThm;
```

9.5 A proof of the Schroëder-Bernstein theorem

The following is a type-theoretic proof of Cantor-Schroeder-Bernstein's theorem given by Hofmann following an idea of J.W. Degen.

```
Init XCC;Logic;

(* Sets *)
[T|Type]; [set=T->Prop]; [subset[X,Y:set]={x:T}(X x)->(Y x)];
[seteq[X,Y:set]=and (subset X Y)(subset Y X)];
[union[X,Y:set]:set=[x:T] or  (X x)(Y x)];
```

```
[inter[X,Y:set]:set=[x:T] and (X x)(Y x)];
[compl[X:set]:set=[x:T]not (X x)];
[setminus[A,B:set]:set = inter A (compl B)];

Goal {A,U,V:set}(subset U V)->subset (setminus A V)(setminus A U);
Intros ___H x H1; Refine pair; Refine fst H1;
Intros H2; Refine snd H1; Refine H; Refine H2;
Save setminus_antitone;

(* Functions *)
[A,B:set];
[total[f:T->T->Prop]={a:T}(A a)->Ex[b:T]and (B b)(f a b)];
[unique[f:T->T->Prop]=
 {a,b1,b2:T}(A a)->(B b1)->(B b2)->(f a b1)->(f a b2)-> Q b1 b2];
[fun[f:T->T->Prop]=and (total f)(unique f)];
[injective[f:T->T->Prop] =
 and (fun f) ({a1,a2,b:T}(A a1)->(A a2)->(B b)->(f a1 b)->(f a2 b)->Q a1 a2)];
[surjective[f:T->T->Prop] =
 and (fun f) ({b:T} (B b) -> Ex [a:T] and (A a) (f a b))];
[im[f:T->T->Prop][U:set]:set=[x:T]and (B x) (Ex[y:T]and (U y) (f y x))];

Goal {f:T->T->Prop}{U,V:set}(subset U V)->(subset V A)->
     subset (im f U)(im f V);
Intros ___ UinV VinA x xinimU; Refine snd xinimU;
Intros finvx; Intros H; Refine pair; Refine fst xinimU; Refine ExIntro;
Refine finvx; Refine pair; Refine UinV; Refine fst H; Refine snd H;
Save im_monotone;
Discharge A;

(* Operators *)
[C:set]; [m:set->set];
[closed={X:set}(subset X C)->subset (m X) C];
[monotone={X,Y:set}(subset X Y)->(subset Y C)->(subset (m X) (m Y))];
[antitone={X,Y:set}(subset X Y)->(subset Y C)->(subset (m Y)(m X))];
Discharge C;
```

```
(* Proof of Tarski's thm for the powerset lattice *)
(* We show that every monotone operator mapping subsets of C to subsets of *)
(* C has a fixpoint which itself is a subset of C *)

[C:set]; [m:set->set]; [cl:closed C m]; [mon:monotone C m];
(*** The `set' of all postfixed points ***)
[G:set->Prop=[Y:set]and (subset Y C)(subset Y (m Y))];
(*** The fixpoint is computed as the union of G ***)
[F:set=[x:T]Ex[Z:set]and (G Z)(Z x)];

Goal subset F C;
Intros f; Intros H; Refine H;
Intros Z; Intros H1; Refine fst(fst H1); Refine snd H1;
Save FinC;

Goal subset F (m F);
Intros f; Intros H; Refine H;
Intros Z; Intros H1; Claim subset (m Z)(m F); Refine ?+1;
Refine snd (fst H1); Refine snd H1;
(* Proving the claim: *)
Refine mon; Intros z; Intros Zz; Refine ExIntro; Refine Z;
Refine pair; Refine fst H1; Refine Zz; Refine FinC;
Save FinmF;

Goal subset (m F) F;
Intros mf; Intros H; Refine ExIntro; Refine m F;
Refine pair; Refine pair; Refine cl; Refine FinC;  Refine mon; Refine FinmF;
Refine cl; Refine FinC; Immed;
Save mFinF;

Goal seteq F (m F);
Refine pair;
Refine FinmF;
Refine mFinF;
Save Ffix;
```

```
Discharge C;

(* Proof of Cantor-Schroeder-Bernstein *)
[A,B:set]; [X,Y:T->T->Prop]; [injX:injective A B X]; [injY:injective B A Y];
[totX=fst(fst injX)]; [uniX=snd(fst injX)];
[totY=fst(fst injY)]; [uniY=snd(fst injY)];

(*** We'll use the gfp of the following operator ***)
[Z:set->set=[U:set]setminus A (im A Y (setminus B (im B X U)))];

Goal closed A Z;
Intros L; Intros H; Intros zl; Intros H1; Refine fst H1;
Save Z_closed;

Goal monotone A Z;
Intros U V UinV VinA;
Refine setminus_antitone; Refine im_monotone B;
Refine setminus_antitone; Refine im_monotone A; Immed;
Intros x H; Refine fst H;
Save Z_monotone;

[FZ=F A Z]; (** FZ is the lfp of Z ***)
[FZfix=Ffix A Z Z_closed Z_monotone];

(*** We define Bij to be X on FZ and Y^-1 on A\FZ ***)
[Bij[a,b:T]=and ( (FZ a)->X a b) ( (setminus A FZ a)->Y b a)];

(** To prove that this is in fact a bijection, we need a classical principle *)

[fully_decidable[A:set]={U:set}(subset U A)->
       subset A (union U (setminus A U))];
[A_dec : fully_decidable A];
[B_dec : fully_decidable B];

(** In fact A_dec will be called for U=FZ and U=Y''(B\X''FZ)  **)
(** ('' denotes image) B_dec will be called for U=X''FZ ***)
```

```
Goal {U:set}(subset U A)->subset (setminus A (setminus A U)) U;
Intros U UinA x H; Refine A_dec U UinA x; Refine fst H; Refine I;
Intros H1; Refine snd H; Immed;
Save double_neg;

(** The following two lemmas state that (A\FZ) = Y''(B\(X''FZ) which
    follows from the definiton of FZ and the involutivity of A\? (deduced
    from the classical assumption) **)

Goal subset (setminus A FZ) (im A Y (setminus B (im B X FZ)));
Intros x; Refine compose; Next +1; Refine double_neg;
Intros x1 H;Refine fst H; Refine setminus_antitone; Refine snd FZfix;
Save lemma1;

Goal subset  (im A Y (setminus B (im B X FZ)))(setminus A FZ);
Intros x; Intros H; Refine setminus_antitone;
Refine setminus A (im A Y (setminus B (im B X FZ)));
Refine fst FZfix; Refine pair; Refine fst H;
Intros H1; Refine snd H1; Refine H;
Save lemma2;

Goal subset FZ A;
Intros f FZf; Refine fst (fst FZfix f FZf);
Save FZinA;

(** Bij is a total relation **)
Goal total A B Bij;
Intros a Aa; Refine A_dec FZ; Refine FZinA; Refine a; Refine Aa;
Intros FZa; Refine totX a; Refine Aa; Intros Xa HXa; Refine ExIntro;Refine Xa;
Refine pair; Refine fst HXa; Refine pair;
Intros _; Refine snd HXa;
Intros H1;Refine snd H1; Refine FZa;
Intros AmFZa; Refine snd (lemma1 a AmFZa);
Intros Yinva; Intros HYinva; Refine ExIntro; Refine Yinva;
Refine pair; Refine fst (fst HYinva);
Refine pair; Intros FZa; Refine snd AmFZa; Refine FZa;
Intros _;Refine snd HYinva;
Save totBij;

(** Bij is functional **)
Goal unique A B Bij;
Intros a b1 b2 Aa Bb1 Bb2 H1 H2;
Refine A_dec FZ; Refine FZinA; Refine a; Refine Aa; Intros FZa; Refine uniX;
Refine a;Immed;
Refine fst H1; Refine FZa; Refine fst H2; Refine FZa;
Intros notFZa;  Refine snd injY; Refine a;Immed;
Refine snd H1;Immed; Refine snd H2;Immed;
Save uniBij;

(** Bij is a function **)
Goal fun A B Bij;
Refine pair; Refine totBij;Refine uniBij;
Save funBij;

(** Injectivity **)
Goal injective A B Bij;
Refine pair; Refine funBij;
Intros ___Aa1 Aa2 Bb H1 H2; Refine A_dec FZ FZinA a1 Aa1;
Intros FZa1; Refine A_dec FZ FZinA a2 Aa2;Intros FZa2;
(** 1st case a1,a2 both in FZ **)
Refine snd injX a1 a2 b;Immed;
Refine fst H1;Immed; Refine fst H2;Immed;
(** 2nd case: a1 in FZ, a2 in A\FZ **)
Intros notFZa2; Claim im B X FZ b; Claim setminus B (im B X FZ) b;
(*** Reasoning by contradiction **)
Refine snd ?+2;Refine ?+1;
(*** Proving that b in X''FZ **)
Refine pair;Immed;Refine ExIntro;Refine a1;Refine pair;Immed;
Refine fst H1;Immed;
(*** Proving that b in B\X''FZ ***)
Claim im A Y (setminus B (im B X FZ)) a2;
```

```
Refine snd ?+1;Intros b' Hb';
Claim Q b' b;
Refine ?+1 [b:T]setminus B (im B X FZ) b; Refine fst Hb';
(*** Proving b=b' ***)
Refine snd injY; Refine a2;Immed; Refine fst (fst Hb');Refine snd Hb';
Refine snd H2;Refine notFZa2;
(*** Proving a2 in Y''(B\X''FZ) ***)
Refine lemma1;Immed;
(** Done **)
Intros notFZa1;
Refine A_dec FZ FZinA a2 Aa2;Intros FZa2;
(** 3rd case: a1 in A\FZ, a2 in FZ works like
the 2nd case with a1 a2 interchanged**)
Claim im B X FZ b;Claim setminus B (im B X FZ) b;
Refine snd ?+2;Refine ?+1;
Refine pair;Immed;Refine ExIntro;Refine a2;Refine pair;Immed;
Refine fst H2;Immed;
Claim im A Y (setminus B (im B X FZ)) a1;
Refine snd ?+1;Intros b' Hb';
Claim Q b' b;
Refine ?+1 [b:T]setminus B (im B X FZ) b;
Refine fst Hb';Refine snd injY;Refine a1;Immed;
Refine fst (fst Hb');Refine snd Hb';Refine snd H1;
Refine notFZa1;
Refine lemma1;Immed;

(** 4th case: a1,a2 both in A\FZ **)
Intros notFZa2;
Refine uniY; Refine b;Immed; Refine snd H1;Immed; Refine snd H2;Immed;
Save injBij;




(** Surjectivity **)
Goal surjective A B Bij;
Refine pair;Refine funBij;
```

```
Intros b Bb; Refine B_dec (im B X FZ) ? b Bb;
Intros b1 H;Refine fst H;
(** 1st case: b in X''FZ **)
Intros imXb;
Refine snd imXb; (** We use the X-inverse of b **)
Intros a Ha; Refine ExIntro;Refine a;
Refine pair; Refine FZinA;Immed;
Refine fst Ha;
Refine pair; Intros FZa;Refine snd Ha;
Intros notFZa;Refine snd notFZa (fst Ha);
(** 2nd case: b in B\X''FZ **)
Intros notimXb;Refine totY b; (** We use a := Y(b) **)
Refine Bb;Intros a Ha;Refine ExIntro; Refine a;
Refine pair;Refine fst Ha;
Refine pair;
Intros FZa; (** In fact we have a in A\FZ **)
Claim setminus A FZ a;
Refine snd ?+1 FZa;
(** Proving the claim **)
Refine lemma2;
Refine pair; Refine fst Ha;Refine ExIntro;Refine b;
Refine pair; Refine notimXb;Refine snd Ha;
Intros notFZa; Refine snd Ha;
Save surBij;
```

A The Syntax of LEGO Expressions

The syntax of LEGO expressions is given in this appendix. It reflects the current version of the system. We write comments between `(* *)`. `n` stands for an arbitrary natural number, `id` for identifiers and `string` for strings.

```
(* expressions *)
exps  ::= exp | exps ; exp
exp   ::= term | context | command | definition

(* terms *)
term ::= app | term : app
app ::= okterm | app space okterm | app bar okterm
okterm ::= id | Prop | Type(n) | Type |
    { decl } app | okterm -> app |
    [ decl ] app | okterm . okterm  |
    < decl > app | okterm # app  |
    okterm , app | okterm.1 | okterm.2 |
    [ defn ] app | ?n | ?+n | ? | ( term )
id-list ::= id | id-list , id
id_#-list ::= id | # | _ | id_#-list , id | id_#-list , # | id_#-list , _
space   ::=
bar     ::= |

(* declarations, definitions and contexts *)
context ::= decl | context decl
binding ::= [ binding ] | [ defn ]
decl    ::= id-list : app | id-list bar app
defn ::= id context = term

(* commands *)
command ::= Goal term |
            Intros id_#-list | Intros n id_#-list | Intros +n id_#-list
            Refine term | Refine n term | Refine +n term
            Immed  |
            Next n | Next +n
            Expand id-list |
            Expand VReg id-list | Expand TReg id-list |
            Equiv term | Equiv VReg term | Equiv TReg term |
            Qrepl term |
            Claim term |
            Prf |
            Undo n |
            KillRef |
            Init id |
            Ctxt | Ctxt n |
            Decls | Decls n |
            Forget id |
            Save id |
            Discharge id | DischargeKeep id |
            Freeze id-list | Unfreeze id-list | Unfreeze
            Help |
            Logic |
            Include id | Include string |
            ExportState id | ExportState string |
            Pwd | Cd string |
            line | echo string

(* definitions --- ‘syntactic sugars’ of  [defn]  in forming contexts. *)
definition ::= id = term |
               id context = term |
               id : app = term   |
               id context : app = term
```

B Logical Definitions

The following are some of the logical definitions definable in the calculus of constructions with type hierarchy. They are the definitions obtained in the current context after command `Logic` is evaluated (for ECC).

```
(* logic.ml *)  (** Logical Preliminaries **)

[A,B,C,D|Prop][a:A][b:B][c:C][d:D][T,S,U|Type];

(* cut *)
[cut = [a:A][h:A->B]h a : A->(A->B)->B];

(* Some Combinators *)
[I [t:T] = t : T]
[compose [f:S->U][g:T->S] = [x:T]f (g x) : T->U]
[permute [f:T->S->U] = [s:S][t:T]f t s : S->T->U];
DischargeKeep A;

(* Conjunction, Disjunction and Negation *)
[and [A,B:Prop] = {C|Prop}(A->B->C)->C : Prop]
[or  [A,B:Prop] = {C|Prop}(A->C)->(B->C)->C : Prop]
(* Introduction Rules *)
[pair = [C|Prop][h:A->B->C](h a b) : and A B]
[inl = [C|Prop][h:A->C][_:B->C]h a : or A B]
[inr = [C|Prop][_:A->C][h:B->C]h b : or A B]
(* Elimination Rules - 'and' & 'or' are their own elim rules *)
[fst [h:and A B] = h [g:A][_:B]g : A]
[snd [h:and A B] = h [_:A][g:B]g : B]

(* Logical Equivalence *)
[iff [A,B:Prop] = and (A->B) (B->A) : Prop]

(* Negation *)
[absurd = {A:Prop}A]
[not [A:Prop] = A->absurd];
```

```
(* Quantification *)
(* a uniform Pi *)
[All [P:T->Prop] = {x:T}P x : Prop]
(* Existential quantifier *)
[Ex [P:T->Prop] = {B:Prop}({t:T}(P t)->B)->B : Prop]
[ExIntro [P:T->Prop][wit|T][prf:P wit]
 = [B:Prop][gen:{t:T}(P t)->B](gen wit prf) : Ex P]
(* Existential restricted to Prop has a witness *)
[ex [P:A->Prop] = {B:Prop}({a:A}(P a)->B)->B : Prop]
[ex_intro [P:A->Prop][wit|A][prf:P wit]
 = [B:Prop][gen:{a:A}(P a)->B](gen wit prf) : ex P]
[witness [P|A->Prop][p:ex P] = p A [x:A][y:P x]x : A];

(* tuples *)
[and3 [A,B,C:Prop] = {X|Prop}(A->B->C->X)->X : Prop]
[pair3 = [X|Prop][h:A->B->C->X]h a b c : and3 A B C]
[and3_out1 [p:and3 A B C] = p [a:A][_:B][_:C]a : A]
[and3_out2 [p:and3 A B C] = p [_:A][b:B][_:C]b : B]
[and3_out3 [p:and3 A B C] = p [_:A][_:B][c:C]c : C]
[iff3 [A,B,C:Prop] = and3 (A->B) (B->C) (C->A) : Prop];

(* finite sums *)
[or3 [A,B,C:Prop] = {X|Prop}(A->X)->(B->X)->(C->X)->X : Prop]
[or3_in1 = [X|Prop][h:A->X][_:B->X][_:C->X](h a) : or3 A B C]
[or3_in2 = [X|Prop][_:A->X][h:B->X][_:C->X](h b) : or3 A B C]
[or3_in3 = [X|Prop][_:A->X][_:B->X][h:C->X](h c) : or3 A B C];

(* Relations *)
[R:T->T->Prop]
[refl = {t:T}R t t : Prop]
[sym = {t,u|T}(R t u)->(R u t) : Prop]
[trans = {t,u,v|T}(R t u)->(R u v)->(R t v) : Prop];
Discharge R;
(* families of relations *)
[respect [f:T->S][R:{X|Type}X->X->Prop]
  = {t,u|T}(R t u)->(R (f t) (f u)) : Prop];
```

```
DischargeKeep A;

(* Equality *)
[Q [x,y:T] = {P:T->Prop}(P x)->(P y) : Prop]
[Q_refl = [t:T][P:T->Prop][h:P t]h : refl Q]
[Q_sym = [t,u|T][g:Q t u]g ([x:T]Q x t) (Q_refl t) : sym Q]
[Q_trans : trans Q
  = [t,u,v|T][p:Q t u][q:Q u v][P:T->Prop]compose (q P) (p P)];
DischargeKeep A;
(* application respects equality; a substitution property *)
[Q_resp [f:T->S] : respect f Q
  = [t,u|T][h:Q t u]h ([z:T]Q (f t) (f z)) (Q_refl (f t))];
Discharge A;";


val logicPreludePCC = "
(* logic.ml *)  (** Logical Preliminaries **)

[A,B,C,D|Prop][a:A][b:B][c:C][d:D][T,S,U|Prop];

(* cut *)
[cut = [a:A][h:A->B]h a : A->(A->B)->B];

(* Some Combinators *)
[I [t:T] = t : T]
[compose [f:S->U][g:T->S] = [x:T]f (g x) : T->U]
[permute [f:T->S->U] = [s:S][t:T]f t s : S->T->U];
DischargeKeep A;

(* Conjunction, Disjunction and Negation *)
[and [A,B:Prop] = {C|Prop}(A->B->C)->C : Prop]
[or  [A,B:Prop] = {C|Prop}(A->C)->(B->C)->C : Prop]
(* Introduction Rules *)
[pair = [C|Prop][h:A->B->C](h a b) : and A B]
[inl = [C|Prop][h:A->C][_:B->C]h a : or A B]
[inr = [C|Prop][_:A->C][h:B->C]h b : or A B]
```

```
(* Elimination Rules - 'and' & 'or' are their own elim rules *)
[fst [h:and A B] = h [g:A][_:B]g : A]
[snd [h:and A B] = h [_:A][g:B]g : B]

(* Logical Equivalence *)
[iff [A,B:Prop] = and (A->B) (B->A) : Prop]

(* Negation *)
[absurd = {A:Prop}A]
[not [A:Prop] = A->absurd];

(* Quantification *)
(* a uniform Pi *)
[All [P:T->Prop] = {x:T}P x : Prop]
(* Existential quantifier *)
[Ex [P:T->Prop] = {B:Prop}({t:T}(P t)->B)->B : Prop]
[ExIntro [P:T->Prop][wit|T][prf:P wit]
 = [B:Prop][gen:{t:T}(P t)->B](gen wit prf) : Ex P]
(* Existential restricted to Prop has a witness *)
[ex [P:A->Prop] = {B:Prop}({a:A}(P a)->B)->B : Prop]
[ex_intro [P:A->Prop][wit|A][prf:P wit]
 = [B:Prop][gen:{a:A}(P a)->B](gen wit prf) : ex P]
[witness [P|A->Prop][p:ex P] = p A [x:A][y:P x]x : A];

(* tuples *)
[and3 [A,B,C:Prop] = {X|Prop}(A->B->C->X)->X : Prop]
[pair3 = [X|Prop][h:A->B->C->X]h a b c : and3 A B C]
[and3_out1 [p:and3 A B C] = p [a:A][_:B][_:C]a : A]
[and3_out2 [p:and3 A B C] = p [_:A][b:B][_:C]b : B]
[and3_out3 [p:and3 A B C] = p [_:A][_:B][c:C]c : C]
[iff3 [A,B,C:Prop] = and3 (A->B) (B->C) (C->A) : Prop];

(* finite sums *)
[or3 [A,B,C:Prop] = {X|Prop}(A->X)->(B->X)->(C->X)->X : Prop]
[or3_in1 = [X|Prop][h:A->X][_:B->X][_:C->X](h a) : or3 A B C]
[or3_in2 = [X|Prop][_:A->X][h:B->X][_:C->X](h b) : or3 A B C]
```

```
[or3_in3 = [X|Prop][_:A->X][_:B->X][h:C->X](h c) : or3 A B C];

(* Relations *)
[R:T->T->Prop]
[refl = {t:T}R t t : Prop]
[sym = {t,u|T}(R t u)->(R u t) : Prop]
[trans = {t,u,v|T}(R t u)->(R u v)->(R t v) : Prop];
Discharge R;
(* families of relations *)
[respect [f:T->S][R:{X|Prop}X->X->Prop]
  = {t,u|T}(R t u)->(R (f t) (f u)) : Prop];
DischargeKeep A;

(* Equality *)
[Q [x,y:T] = {P:T->Prop}(P x)->(P y) : Prop]
[Q_refl = [t:T][P:T->Prop][h:P t]h : refl Q]
[Q_sym = [t,u|T][g:Q t u]g ([x:T]Q x t) (Q_refl t) : sym Q]
[Q_trans : trans Q
  = [t,u,v|T][p:Q t u][q:Q u v][P:T->Prop]compose (q P) (p P)];
DischargeKeep A;
(* application respects equality; a substitution property *)
[Q_resp [f:T->S] : respect f Q
  = [t,u|T][h:Q t u]h ([z:T]Q (f t) (f z)) (Q_refl (f t))];
Discharge A;
```

C The ML Level of LEGO

LEGO uses ML as its meta language. One may enter the ML level of LEGO system as follows:

```
shell> legoML
Standard ML with LEGO (.. it's fun to do constructions)
-
```

where '-' is the ML prompt. To enter lego state (see section 2.2) from the top level of ML, you use the ML function `lego : unit -> unit` (at the top level of ML, a semicolon (;) causes the expression just entered to be evaluated):

```
- lego();
Lego>
```

If you loaded LEGO through `legoML`, 'Ctrl-d' (`^D`) will enable you to exit lego state to the top level of ML; at the top level of ML, 'Ctrl-d' (`^D`) will return to UNIX.

```
Lego> ^D
val it = () : unit
- ^D
shell%
```

Therefore, a finer picture of LEGO system may be understood as follows:

$$UNIX\ shell \xrightarrow{\texttt{legoML}} ML \xrightarrow{\texttt{lego();}} lego\ state \rightleftharpoons proof\ state$$

C.1 Environment variable LEGOPATH

There is a UNIX environment variable `LEGOPATH` associated with LEGO system for which more advanced users may need to know. You can set this environment variable (either interactively or by setting it in your `.login` file) to direct the system to search directories when commands like `Include` and `ExportState` are evaluated.

For example, you may set the enrironment variable `LEGOPATH` to `".:~/lego7:/home/usrname/lego/EXAMPLES"`, then, the searching path will be your current directory (specified by '`.`'), the directory `lego7` and the directory `/home/usrname/lego/EXAMPLES`, in the specified order. If the argument of an `Include` (or `ExportState`) command is a relative path (*e.g.*, just a file name), then, the source (or target) file for the command will be the first file (with the extension `.l` optional) found in the directories specified by `LEGOPATH`.

The default value of `LEGOPATH` is '`.`', *i.e.*, the current directory.

C.2 ML functions legos and legof

At the top level of ML, the ML function `legos : string -> unit` takes a string (viewed as an lego expression) and evaluates the expression. For example,

```
- legos"Init XCC";
Extended CC: Initial State!
val it = () : unit
-
```

The ML function `legof : string -> unit` takes a string (viewed as a UNIX path) as argument. It has the same effect (at the ML level) as the command `Include` has (at the level of lego state). For example, if file `semigroup` is in the current directory and `LEGOPATH` is set as its default,

```
- legof"semigroup";
(* opening file ./semigroup.l *)
defn  Sig_SG = <A:Type(0)>A->A->A
      Sig_SG : Type(1)
... ... ...
(* closing file ./semigroup.l *)
val it = () : unit
-
```

References

[AHMP92] A.Avron, F.Honsell, I.Mason, and R. Pollack, *Using Typed Lambda Calculus to Implement Formal Systems on a Machine,* To appear in J. of Automated Reasoning.

[BM91] R. Burstall and J. McKinna, *Deliverables: an Approach to Program Development in the Calculus of Constructions,* ECS-LFCS-91-133, Dept. of Computer Science, Edinburgh University, 1991.

[Bur89a] R. Burstall, *An Approach to Program Specification and Development in Constructions*, Talk given in Workshop on Programming Logic, Bastad, Sweden, May 1989.

[Bur89b] R. Burstall, *'Computer Assisted Proof for Mathematics: an introduction, using the LEGO proof system',* Proc. the Institute for Applied Mathematics, 'The Revolution in Mathematics Caused by Computing', Brighton Polytechnic.

[CH85] Th.Coquand and G.Huet, *'Constructions: a Higher-order Proof System for Mechanizing Mathematics',* EUROCAL'85.

[CH88] Th.Coquand and G.Huet, *'The Calculus of Constructions',* Information and Computation, 2/3, vol. 76.

[Chu40] A. Church, *'A Formulation of the Simple Theory of Types',* J. Symbolic Logic 5 (1).

[Col91] A. Coleman, *Simulating VDM in Lego*, MSc thesis, Dept of Computer Science, University of Edinburgh, 1991.

[Coq86] Th.Coquand, *'An Analysis of Girard's Paradox',* LICS'86.

[CPM90] Th. Coquand and Ch. Paulin-Mohring, *'Inductively defined types'*, Lecture Notes in Computer Science, 417, 1990.

[Dyb91] P. Dybjer, *Inductive sets and families in Martin-Löf's type theory,* In G. Huet and G. Plotkin, editors, Logical Frameworks, Cambridge Univ. Press, 1991.

[HHP87] R.Harper, F.Honsell and G.Plotkin, *'A Framework for defining Logics',* LICS'87.

[HMM86] R. Harper, D. MacQueen and R. Milner, *Standard ML,* LFCS Report ECS-LFCS-86-2, Dept. of Computer Science, Univ. of Edinburgh.

[Hof91] M. Hofmann, *Verifikation von ML-Programmen mit dem Beweisprüfer Lego,* MSc thesis, Diplomarbeit an der Universität Erlangen Nürnberg, 1991.

[HP91] R. Harper and R. Pollack, *'Universe Polymorphism',* Theoretical Computer Science, 89, 1991.

[Hue87] G. Huet, *Extending the Calculus of Constructions with Type:Type,* manuscript. April 1987.

[Jon91] C. Jones, *Completing the Rationals and Metric Spaces in LEGO,* Proceedings of the Second Annual Workshop on Logical Frameworks, Edinburgh, 1991.

[LPT89] Z. Luo, R. Pollack and P. Taylor. *How to Use LEGO: a preliminary user's manual.* LFCS Technical Notes LFCS-TN-27, Dept. of Computer Science, Edinburgh Univ., 1989.

[Luo89] Z. Luo, '**ECC***: an Extended Calculus of Constructions',* Proc. of the Fourth Ann. Symp. on Logic in Computer Science, June 1989, Asilomar, California, U.S.A.

[Luo90] Z. Luo, *An Extended Calculus of Constructions,* PhD thesis, University of Edinburgh, 1990. Also as report CST-65-90/ECS-LFCS-90-118, Dept. of Computer Science, Edinburgh University, 1990.

[Luo91a] Z. Luo, *'A Higher-order Calculus and Theory Abstraction',* Information and Computation, 90(1), 1991.

[Luo91b] Z. Luo, *Program Specification and Data Refinement in Type Theory.* Proc. of the Inter. Joint Conf. on the Theory and Practice of Software Development (TAPSOFT'91, LNCS 493), Brighton, 1991. Also as LFCS report ECS-LFCS-91-131, Dept. of Computer Science, Edinburgh University.

[Luo91c] Z. Luo, *A Problem of Adequacy: conservativity of calculus of constructions over higher-order logic.* LFCS report ECS-LFCS-90-121, Dept. of Computer Science, Edinburgh University.

[Luo92] Z. Luo, *'A unifying theory of dependent types: the schematic approach',* To appear in Proc. of Symp. on Logical Foundations of Computer Science (Logic at Tver), Tver, 1992. Also as LFCS report ECS-LFCS-92-202, Dept of Computer Science, Univ. of Edinburgh.

[Mah91] S. Maharaj, *Implementing Z in Lego,* MSc thesis, Dept of Computer Science, University of Edinburgh, 1991.

[McK92] J. McKinna, *Deliverables: a categorical approach to program development in type theory,* PhD thesis in preparation.

[Pol88] R. Pollack, *The Theory of LEGO,* Manuscript. 1988.

[Pol90] R. Pollack, *'Implicit Syntax',* Preliminary Proc. of the 1st Workshop on Logical Frameworks, Antibes, 1990.

[Pol92] R. Pollack, The Theory of Lego. PhD thesis in preparation.

[Tak75] G. Takeuti, *Proof Theory,* Stud. Logic 81.

[Tay88] P. Taylor, *Using Constructions as a Meta Language,* LFCS report ECS-LFCS-88-70, Dept. of Computer Science, Univ. of Edinburgh.

[Tay89] P. Taylor, *Playing with LEGO: some examples of developing mathematics in the calculus of constructions,* LFCS report ECS-LFCS-89-89, Dept. of Computer Science, Univ. of Edinburgh.