Formal Development of Functional Programs in Type Theory — A Case Study

Martin Hofmann
Department of Computer Science, University of Edinburgh
JCMB, KB, Mayfield Rd., Edinburgh EH9 3JZ, Scotland

July 31, 1992

Abstract

Type systems like the Extended Calculus of Constructions (ECC) [17] have been proposed as an environment in which to develop both modular programs and their correctness proofs. This work describes a case study in this area which uses the implementation Lego by R. Pollack of the ECC. After introducing the main concepts of type-theoretic specification and verification we describe a machine-checked version of the symbol table example given in [24]. Particular attention is paid to a detailed description of the actual interaction with Lego as a program development system, so as to provide a basis for further research on this topic. The case study is a shortened version of the author's *Diplomarbeit* (Master's thesis) [14]

1 Introduction

Recently several specification languages supporting the formal program development process have been designed. They allow to formalise the specification of a computer program using some logical formalism such as first order equational logic. A well known shortcoming of most of these specification languages, referred to as "the semantic gap", is their inability to deal with both specifications *and* programs. The language *Extended ML* [25] is an important exception to this. It allows to refine a given specification to an actual program via several steps involving both code and specifications of the remaining parts. The correctness proofs, however, remain unformalised in Extended ML; they must — at least at the current stage of development [27] — be done by paper and pencil outside the system.

In some sense orthogonal to specification languages stands traditional program verification like the Floyd-Hoare approach or the Boyer-Moore theorem prover. Here a previously given program is formally verified in one step. The program development process is not treated in these systems. Moreover when programs become larger, automated or one-step verification becomes difficult to manage.

The aim of this work is to study a specification formalism based on type theory which unifies these two streams of research in the sense that formal machine-supported verification is combined with methods from program specification and formal program development. Based on the type system ECC [17] and its implementation Lego [19] we describe how to use type theory to develop both a program and a formal proof of its correctness starting from a specification. The method of program development will be the one of Extended ML, i.e. top down and by stepwise refinement, which, however, does not mean that the use of ECC/Lego for program development is restricted to this method. Our main intention is to demonstrate the practical use of type systems like ECC/Lego for actual program development and verification, in order to provide a basis for future research in this area. For the sake of completeness we also treat some of the theoretical underpinnings of our work without claiming originality here.

Before describing our work in more detail, we mention two major differences to related approaches, to prevent possible misunderstandings. Firstly we intend neither automated verification, nor automatic program derivation from specifications. Both program and correctness proof have to be supplied by the user; the system only performs the context management and proof/type checking. Hence we are free to use complex logical systems such as higher-order logic as specification formalisms. Secondly we insist on (simply) decidable proof checking, which allows the use of user-defined tactics[1] without loss of reliability. (The idea is that every tactic, however complicated it might be, finally yields an explicit proof term which is checked by a simple and reliable system.)

The rest of the paper is organised as follows. First we give a brief introduction to the type system ECC and the proof checker Lego. Secondly we describe the use of ECC as a programming and specification language; we separately treat the underlying programming language, the logic used for specifications, and the module facilities. In the last section we describe our (Lego) version of a formal development of a modular implementation of a symbol table [24] which both illustrates the use of ECC as a specification language and allows a comparison with other specification/verification formalisms. The description of the example is self-contained but will be best understood if the Lego code is actually executed on a machine. To that end the whole of the definitions and proofs have been made available by anonymous ftp or e-mail (See Appendix).

2 ECC and Lego

The key to the unified treatment of programs and (correctness) proofs is the propositions (viz. specifications) as types paradigm initially observed by Kolmogorov and Heyting and subsequently exploited by Martin-Löf [21]. While ECC does not completely identify types and propositions it still keeps both in the same formalism: propositions are a special kind of types, proofs are their members.

[1] At th current stage of development Lego does not support user-defined tactics wet. It contains, however, several built-in tactics which all produce explicit proof terms.

From an abstract point of view ECC is a formal system for deriving judgments of the form

$$\Gamma \vdash t : T$$

where Γ is a context (of assumptions and definitions), t is a term and T is its type. In the context of program development there is, however, a variety of interpretations of such judgments. If T is a proposition, which we think of as the *type of its proofs* then the judgment reads "t is a proof of T". If T is a programming language type, such as `nat`, `nat->bool`, `list(E)->nat` — these types are called *small types* — then the judgment states that t is a *value* of that type, e.g. the constant `zero:nat`, the function `is_odd:nat->bool`, or the length function on lists. Finally, if T is a signature, then the judgment denotes matching of structures, which — since our signatures in the sense of SML modules [12] will contain both types and predicates (in Extended ML these are called *axioms*) — means that t is a structure containing values of the indicated types which is provably correct w.r.t. the predicates stated in the signature.

The types of Lego/ECC have a type themselves which is called a *universe*. Propositions and small types (`nat`, `bool`) belong to the same universe, which for historical reasons is denoted `Prop` in Lego. The universe of signatures and specifications (see section 5) is denoted `Type`.[2] As a consequence of Girard's paradox [8] signatures and small types cannot live in the same universe. The existence of these universes allows the typing of polymorphic functions and quantification over all predicates. E.g. the polymorphic `append` function has type

```
{E|Prop}(list E) -> (list E) -> (list E)
```

This type needs some explanation. The curly brackets are notation for dependent function space; hence `append` works for lists over an arbitrary type `E` of entries. The vertical bar in place of the usual colon (`{E|Prop}`) indicates *implicit polymorphism*: if `l1`, `l2` are both lists over `E` (they both have type `list E`), then their concatenation is written `(append l1 l2)` — the first argument of `append`, namely the entry type, need not be given explicitly.

2.1 The proof checker Lego

Here we briefly sketch some features of Lego; more detailed examples of its use will be given throughout the rest of the paper, mostly in section 6. For a more gentle introduction to Lego the reader is referred to [4, 19, 28].

The heart of the Lego system is a type checker which, given a typing judgment, decides whether it is correct. Moreover it manages a context which we can enlarge by assumptions written in square brackets: the command

```
[A:Prop];
```
[3]

[2] This is an oversimplified presentation of ECC's type structure. The interested reader is referred to [17].

Datatype	Lego notation	Definition
Unit type	`unit`	`{C|Prop} C -> C`
Natural numbers	`nat`	`{C|Prop} C -> (C->C) ->C`
Lists (`E list`)	`(list E)`	`{C|Prop} C -> (E->C->C) -> C`
Cartesian product	`(PROD A B)`	`{C|Prop} (A->B->C) -> C`
Disjoint union	`(SUM A B)`	`{C|Prop} (A->C) -> (B->C) -> C`
Function space	`A -> B`	*abbreviating* `{a:A}B`

Figure 1: Basic datatypes in Lego

adds a variable `A` of type `Prop` to the context. We also can put definitions in the context: the command

```
[idA[a:A] = a];
```

declares that henceforth `idA` is an abbreviation for the identity function on `A`. Finally, and this is the actual power of Lego, it supports the task of finding an inhabitant of a certain type (called "goal") by an involved refinement mechanism. If `T` is a type (a proposition, a signature, a specification) then the command

```
Goal T;
```

invokes the interactive refinement mode; once we have constructed an inhabitant of `T` — how this can be done is shown e.g. on page 9 — Lego prints

```
*** QED ***
```

We can then save the proof, resp. the value, according to what the intention of `T` is, using

```
Save prf;
```

This has the same result as if we had made the definition

```
[prf:T = ...];
```

where `...` stands for the proof term.

3 The programming language

We briefly describe the programming language or rather the subset of ECC which we use for programming. By means of various definitions it becomes quite close to an explicitly typed version of core ML. The most fundamental difference is the lack of partial functions, which we view as being not necessarily restrictive. This lack is due to the fact that Lego/ECC is a strongly normalising system in which all programs terminate. The main advantage of this is that a program–proof pair can

[3] Henceforth we use `typewriter` font for actual Lego notation, which can be typed in directly.

be checked against its specification within a finite amount of time. Cf. the idea of *deliverables* [6]. Furthermore we do not treat exceptions or references; this, however, seems achievable in future.

Since ECC has no built-in types at all, we must define the basic datatypes like `nat`, `bool`, `list`, etc.. There are several ways to do this; we have chosen the so–called "impredicative coding" which represents datatypes by their polymorphic iterators (see figure 1). We do not describe this encoding here, the interested reader is referred to [2, 15]. What is important is that it allows us to define usual inductive types together with constructors and a suitable recursion principle. So we have a type `nat` with the constructors `zero:nat`, `succ:nat->nat` and a recursor `natrec` of type

```
{R|Prop}R -> (nat -> R -> R) -> nat -> R
```

This polymorphic function is used to define functions from `nat` to some other type `R` by primitive recursion. It is best understood in terms of ML pattern matching. The ML declaration

```
val rec f = fn
   zero => g
   |succ n => h n (f n)
```

translates to

```
[f=natrec g h];
```

where `f` and `g` must have the types `R` and `(nat->R->R)` respectively. Lego will acknowledge this definition by printing

```
defn f = natrec g h
     f :  nat->R
```

By reduction we can run actual computations like

```
Normal (natrec g h (succ (succ (zero))));
value = h ([C|Prop][z:C][s:C->C]s z) (h ([C|Prop][z:C][_:C->C]z) g)
type = R
```

The expressions `([C|Prop][z:C][s:C->C]s z)` and `(h ([C|Prop][z:C][_:C->C]z)` are the normal forms for the numerals `(succ zero)` and `zero`, resp. . Unless explicitly asked by the `Normal` statement Lego performs reductions only if necessary to solve a goal, hence the number of reduction steps is kept small.

Other data types are the truth values `bool` and finite lists `list` where the constructors and recursors have the following types:

```
[true,false:bool]
[if:{R|Prop}bool -> R -> R -> R]

[nil:{E:Prop}list E];
[cons:{E|Prop}E -> (list E) -> (list E)];
[listrec:{E|Prop}{R|Prop}R -> ((list E) -> E -> R -> R) -> (listE) -> R]
```

which again allow the translation of the corresponding ML patterns. Notice that all type parameters except the one of `nil` are implicit. The latter has to be instantiated explicitly since the appropriate type cannot be inferred from the other arguments here.

As an illustration we give some simple function definitions. The square brackets inside a term denote λ-abstraction and can be used on either side of the defining '`=`' sign.

```
[orelse:bool->bool->bool =
   [b1,b2:bool] if b1 true b2];

[andalso:bool->bool->bool =
   [b1,b2:bool] if b1 b2 false];

[isnil:{E|Prop}(list E)->bool =
   listrec true ([_:list E][_:E][_:bool]false];

[eq_nat:nat->nat->bool = natrec
   (natrec true ([_:nat][_:bool]false))
   ([n:nat][eqn:nat->bool]natrec false ([m:nat][_:bool]eqn m))];
```

Observe the shorthand for subsequent λ-abstractions of the same type like in `[b1,b2:bool]...` and the "dummy" notation for parameters which are not actually used (`[_:E][_:bool]...`). The last definition (`eq_nat`) implements an equality function on the natural numbers. It is slightly involved since its result type `R` is a function type (`nat->bool`). In the Lego code (see Appendix) a correctness proofs for this function is given.

As mentioned in the beginning of this section all programs written using these recursors terminate. *A fortiori* we cannot express all computable functions in this framework, and no general recursion is available. Since, however, the result type `R` of a recursor may be a function type (cf. `eq_nat` above), we can program more than just the (first order) primitive recursive functions. To be precise, all those functions whose termination is provable in higher–order arithmetic are representable [8]. Since a termination proof is part of every verification this does not seem to be an overly severe restriction. Notice also that a partial function the domain of which is decidable can be simulated by a total one with some suitably extended codomain. On the other hand a partial function with undecidable domain is quite useless, at least if we want to prove something about it. One might argue that functions like interpreters for programming languages are useful and yet their domain is undecidable. But the only way to specify such a function and to prove it correct is by introducing a "one–step evaluation" function which again is amenable to our approach.

A quite different issue, however, is that even if we can represent the underlying function of a recursive algorithm, the recursive algorithm might be more efficient or more intuitive than its primitive recursive simulation. Typical examples are the Euclidean algorithm or sorting procedures like QuickSort or MergeSort. If we want to verify one of these algorithms, it does not help to have just an extensionally equivalent primitive recursive function; we want to reason about the recursive algorithm directly. To cope with this problem we have defined a combinator `recurse` (using `natrec`) which, given a recursion scheme and a termination proof, computes a solution of the scheme. Here we only consider termination proofs by induction along ω. It is, however, possible to give generalisations to arbitrary well–founded orderings, as long as their wellfoundedness can be shown constructively within the system. The combinator has the type

`{A,B|Prop}`	Types of argument and result
`    (A->bool)->`	Termination condition
`    (A->B)->`	Base case
`    (A->(A->B)->B)->`	Actual recursion scheme
`    (A->nat)->`	Complexity measure
`        A->B`	The resulting function

Given values `end`, `base`, `scheme`, `w` of the appropriate types then provided the measure `w` is actually reduced by the recursion scheme (how this is formulated is shown in section 6) the function

```
[f = (recurse end base scheme)]
```

provably satisfies the recursive equations, i.e. for all `a` in `A`

```
(f a) =4(if (end a) (base a) (scheme a f))
```

and a proof principle comparable to Scott induction, the form of which is shown in section 6, holds. The Lego proofs of these two are called `recurse_law`, and `recind` in the Lego code. As an example we show how we can program the Euclidean algorithm using this method. We write down the recursive definition, where we assume the elementary functions and type constructors `PROD`, `fst`, `snd`, `eqzero`, `minus`, `le` (less or equal) to be given (with the obvious meanings):

```
[A=PROD nat nat];                                          the domain of GCD

[end:A->bool=[a:A]orelse (eqzero (fst a))(eqzero (snd a))];      termination

[base:A->bool =
   [a:A][m=fst a][n=snd a]=if (eqzero m) n m];                   base case

[scheme:A->(A->nat)->nat =                          actual recursion scheme
   [a:A][gcd:A->nat][m=fst a][n=snd a]
   if (le m n) (gcd m (minus n m)) (gcd (minus m n) n)];
```

Observe the local definitions `[m=fst b]`, `[n=snd b]` which define `n` and `m` to be the first/second component of `b` within the scope of definition in which they appear. This reflects the ML definition

[4] See next section for the precise meaning of the equality (=) used in an informal way here.

Usual notation	Lego notation	Definition
$A \wedge B$	`(and A B)`	`{C|Prop}(A->B->C) -> C`
$A \vee B$	`(or A B)`	`{C|Prop}(A->C) -> (B->C) -> C`
$A \Rightarrow B$	`A -> B`	`{_:A}B`
$\perp$ (falsity)	`absurd`	`{C:Prop} C`
$\neg A$	`not A`	`A -> absurd`
$A \Leftrightarrow B$	`(iff A B)`	`(and (A->B) (B->A))`
$\forall x \in A.P(x)$	`{x:A} (P x)`	*primitive*
$\exists x \in A.P(x)$	`Ex[a:A]P a` or `Ex P`	`{C|Prop}({a:A}(P a) -> C) -> C`
$x = y \in X$	`(Q x y)`	`{P:X->Prop} (P x) -> (P y)`

Figure 2: Logical connectives in Lego

```
fun GCD(m,n) =
if m=0 orelse n=0
  then (if m=0 then n else m)
  else if m<=n
       then GCD(m,n-m)
       else GCD(m-n,n);
```

Whereas ML can compute the function directly from this definition, we must supply an explicit complexity measure which is reduced by each recursive step:

```
[w:B->nat=[b:B]plus (fst b)(snd b)];
```
complexity measure

and obtain the GCD function by

```
[GCD = recurse end base scheme w];
```

The proof that the recursion scheme actually reduces the complexity measure `w` is not required here but as a premise for the correctness proof of `recurse`, and also for a proof principle similar to Scott induction, which are used to reason about functions defined using `recurse`.[5] In 6.3.3 we show an application of both principles.

4 The logic

The logic in which to express properties of our programs is higher-order intuitionistic logic[6]. As in the case of the data types we must define the usual connectives from universal quantification (`{x:A}B x`) and its special case "implication" (`A ->`

[5] If no such proof would exist, then we could still run the function defined by recursion, but it would not satisfy the recursive equations.

[6] The restriction to intuitionistic logic is necessary if we want to use unrestricted polymorphism, and also since we have propositions and (small) types in the same universe `Prop`. In future we intend to introduce separate universes for types and propositions [1, 9], which would allow us to assume non-constructive principles for propositions without affecting the types. This, however, requires a restriction of polymorphism, as well.

B) which are the only built-in logical operators. These definitions are, however, loaded automatically by the command `Logic;`, together with the corresponding introduction and elimination rules. Using these connectives we can state and prove propositions like

```
Goal {P,Q:Prop}(and P Q) -> P;                                  ∀P,Q.(P ∧ Q) → P
   ?0 :  {P,Q:Prop}(and P Q) -> P;        The indented parts are output by the system
Intros P Q Hyp;
   P : Prop
   Q : Prop
   Hyp :  and P Q
   ?0 :  P
Refine fst Hyp;
   *** QED ***
Save proof;proof;                                 Saving the proof and printing it out
   value = [P,Q:Prop][Hyp:and P Q]fst Hyp                                The proof term
```

This simple proof shows the two main mechanisms of Lego's interactive proof mode: introduction (`Intros` *var*) and refinement (`Refine` *term*). `Intros` applies to universal quantifications and implication and introduces a fresh variable or a hypothesis, respectively. The command `Refine` *term*; uses the supplied term (which may contain previously introduced assumptions and even other subgoals) to prove the current goal. The used premise may be more general than the goal or require other preconditions. In this case Lego tries to find the appropriate instantiation by unification and generates new subgoals corresponding to the missing premises:

```
Goal {A:Prop}{R:A->A->Prop}{x,y:A}({b:A}R x b) -> Ex[a:A]R a y;
                                     ∀A.∀R.∀x,y ∈ A.(∀b ∈ A.R(x,b)) → (∃a ∈ A.R(a,y))
   ?0 :  {A:Prop}{R:A->A->Prop}{x,y:A}({b:A}R x b)->Ex [a:A]R a y)
Intros A R x y H;
   A : Prop
   R : A->A->Prop
   x :  A
   y :  A
   H : {y:A}R x y
   ?1 :  Ex ([x:A]R x y)
Refine ExIntro;
   ?4[7]:  A                                             we have to supply a witness
   ?5 :  R ?4 y                                          and a proof of its adequacy
Next +1;                                                 now ?5 becomes the next goal
Refine H;
   *** QED ***                          both ?4 and H are instantiated by unification.
Save proof;proof;
   value = [A:Prop][R:A->A->Prop][x,y:A][H:b:AR x b]
      ExIntro ([a:A]R a y) (H y)
```

Notice the proof term which can be easily checked without using unification or whatever tactics lead to it.

Higher-order logic also permits the internal definition of equality (figure 2):

[7] Lego does not necessarily generate consecutive goal numbers.

```
[Q[A|Prop][a,b:A]={P:A->Prop} (P a)->(P b)]
```
$a = b \Leftrightarrow_{\texttt{def}} \forall P.\, P(a) \rightarrow P(b)$

according to Leibniz's principle stating that undistinguishable objects should be considered equal. Proofs that this definition satisfies the usual laws are loaded automatically with the `Logic` command. The *use* of `Q`-equalities to substitute equal terms inside a goal is — though actually immediate from the definition — a bit tricky. To see how this can be done consider the following situation:

```
m :  nat                              a natural number
H : Q zero m                          assumption that m equals zero
?1 :  {n:nat} Q (plus n m) n          n + m = n
```

We would now like to substitute `zero` for `m` in the goal `?1` using the hypothesis `H`. In order to see how this can be done let us recall the definition of equality. We have `H:{P:nat->Prop} (P zero) -> (P m)`. This means that in order to prove a property (`P`) of the number `m` it is enough to show it for zero. But our goal is such a property of `m`. So we instantiate the hypothesis with the property the goal expresses and obtain

```
Refine H ([x:nat] {n:nat} Q (plus n x) n);
  ?2 :  {n:nat} Q (plus n zero) n     the substitution has been carried out
```

The newest Lego version contains a tactic `Qrepl` which performs these substitutions automatically. Using it the above goal could have been solved by typing

```
Qrepl (Q_sym H);
```

The proof term itself, however, remain unchanged and not contain any instance of `Qrepl`. See the Lego manual [19] for further information. In order to *establish* equalities we can sometimes use normalisation as in the following case

```
Goal {R|Prop}{r1,r2:R}Q r1 (if true r1 r2);
```

Here due to the impredicative coding of the booleans the expression `if true r1 r2` automatically reduces to `r1` so we can solve this goal simply by

```
Intros R r1 r2;Refine Q_refl;         by reflexivity of Q;
```

Other equalities such as

```
Goal {R|Prop}{r:R}{b:bool}Q r (if b r r);
```

however, do not follow by reduction. To prove such statements we need appropriate induction principles for the data types:

```
[natind:{P:nat->Prop}(P zero)->({n:nat}(P n)->(P (succ n))) ->{n:nat}P n]
```

$$P(0) \to (\forall n.\, P(n) \to P(n+1)) \to \forall n.\, P(n)$$

```
[boolind:{P:bool->Prop}(P true) -> (P false) -> {b:bool}P b]
```

$$P(true) \to P(false) \to \forall b.\, P(b)$$

```
[listind:{E:Prop}{P:(list E)->Prop}
    (P (nil E)) ->
    ({l:list E}{e:E}(P l) -> P (cons e l)) ->
    {l:list E}P l];
```

$$P(\texttt{nil}) \to (\forall l, e.P(l) \to P(\texttt{cons}(e,l)) \to \forall l.P(l)$$

Although consistent ([8]), these induction principles are not provable in the empty context. We therefore have to assume them. In [14] we state a more general axiom from which these principles can be inferred. Using `boolind` we can solve the above goal by

```
Intros R r;
Refine boolind [b:bool]Q r (if b r r);          we suitably instantiate boolind
   ?1 :  Q r (if true r r)
   ?2 :  Q r (if false r r)
Refine Q_refl;Refine Q_refl;                    these goals again follow by reduction
```

To prove negative statements like e.g.

```
Goal {n:nat}not (Q zero (succ n));
```

we must also assume one inequality:

```
[consistency:not (Q true false)]
```

From these assumptions we have derived several lemmas[8] describing the definitions in such a way that it is possible to reason about them without knowing their actual realisation. Some examples can be found in section 6, where also examples for more involved Lego proofs are given.

To end this section we want to mention two advantages of using higher-order logic instead of e.g. first-order equational logic. Firstly in parametrised modules we are able to express without any special constructions that we expect a free algebra as input structure (just by requiring the input structure to satisfy a corresponding induction principle and to come equipped with a recursion scheme). For example we may write

```
signature list =
  sig
    type E
    type L
    val nil:L
    val cons:E->L->L
    val listrec:'a->L->E->'a->'a)->L->'a
    axiom listrec n c nil = n
    axiom listrec n c (cons e l) = c l e (listrec n c l)
    axiom P(nil) → (∀ l, e.P(l) → P(cons e l)) → ∀ l.P(l)
  end
```

[8]See Lego code

(We use Extended ML notation here since we have not yet introduced Lego notation for signatures.) Moreover we can specify that elements of some data structure are generated from a given set of constructors without insisting on the initial algebra (in this example by dropping `listrec`). This corresponds to Extended ML's spantypes [27]. Secondly we believe that some predicates like the property of a list being a permutation of another list are easier to express using inductive predicates which again are a higher–order construction. We remark that for equational and first–order specification languages extra concepts such as data constraints [5] have to be introduced to cope with these issues, which makes them more complicated.

5 Signatures and functors

As in Extended ML we enlarge a signature by axioms which specify logical properties of the structures matching it. Unlike in Extended ML however a *structure* will not only consist of types and functions matching the types given in the signature but also of proof terms proving their correctness w.r.t. the axioms stated in the signature. Type–theoretically we think of a signature as a dependent sum (strong Σ-type) [20, 17] whose members are pairs of programs and proofs. The question whether such a structure matches a given signature (i.e. whether its program part matches the signature in the usual sense and whether the proof terms actually prove its correctness) then boils down to a type checking problem and is done automatically by the Lego system. The list example from above should make things clear. In Lego we write

```
[LIST =
   <E:Prop>
   <L:Prop>
   <nil:L>
   <cons:E->L->L>
   <listrec:{A|Prop}A->(L->E->A->A)->L->A>
   ({n:A}{c:L->E->A->A} Q (listrec n c nil) n) #
   ({n:A}{c:L->E->A->A}{l:L}{e:E}
      Q (listrec n c (cons e l)) (c l e (listrec n c l)) #
   ({P:L->Prop}(P nil)->
      ({l:L}{e:E}(P l) -> (P (cons e l))) ->
         {l:L}P l)
];
```

The sharp (`#`) separates components which do not appear later and hence are unnamed. I.a.w. `A#B` abbreviates `<a:A>B` just as `A->B` is a shorthand for `{a:A}B`. We can now form structures matching this signature by putting the corresponding terms together:

```
[List=(nat,(list nat),nil,cons,itlist,p1,p2,p3):LIST]
```

where `p1, p2, p3` are proof terms which must have been constructed previously. Conversely we can access the components of `List` by projection functions: `List.1` gives the underlying type of elements corresponding to `E` (here `nat`), `List.2.2.2.1` gives `cons`, etc. (similar to LISP's `car` and `cdr` functions). If we have some other signature containing a type `E`:

```
[ENTRY =
   <E:Prop>
   <f:E->nat>
   ...
];
```

then we might want to construct a functor which given `Entry:ENTRY` constructs a list structure over the underlying type. If we just exhibit a "function" of type `ENTRY->LIST` then this could be the intended one, but it might just as well ignore its argument and return a list structure over the empty type, for example. According to our paradigm that type checking should also check correctness we must take a "finer" type for such a functor. In Extended ML this is done using sharing constraints, here:

```
functor FF(Entry:ENTRY) : sig include LIST
                              sharing type E = Entry.E
                          end = ...
```

Among various ways to model this issue type–theoretically, we have chosen the method of explicit parametrisation, which goes back to Pebble [3] (see also [20, 16]). We make the signatures variable in the shared arguments:

```
[LIST' = [E:Prop]
   <L:Prop>
   <nil:L>
   ...
];
[ENTRY' = [E:Prop]
   <f:E->nat>
...
];
```

Notice that such a parametrised signature is a dependent type and has no inhabitants unless instantiated with an appropriate type `E`. Now the type of a functor constructing lists becomes

```
[ENTRYTOLIST = {E:Prop}(ENTRY' E) -> (LIST' E)];
```

Given `[FF:ENTRYTOLIST]` we instantiate it as follows with `[Entry:ENTRY]`: The application `(FF Entry.1 Entry.2)` gives a list structure which now provably has `Entry.1` as entry type. A similar procedure applies for "horizontal" sharing, where two arguments of a functor share a common part. There is, however, a tradeoff; namely we must decide at the design time of a signature which parts we keep inside and which parts are susceptible to being shared. If we adopt Extended ML's top down development strategy, this is a minor problem since the *use* of a signature occurs *before* its actual design. Otherwise we must parametrise as many parts as possible. In future it might also be worthwhile to investigate a signature calculus comparable to the one used in the Z specification language [13] which would allow to change signatures internally.

We have now introduced the use of ECC and Lego as a tool for machine supported specification and verification. We have shown how functional programs, higher–order logic, and modularisation all fit into the same formalism. More involved examples will appear in the next section where we describe our case study based on [24]. We want to point out that we have only introduced those facets of the ECC which appear in the case study. The full ECC is much more extensive, e.g. it allows dependent types and programs depending on proofs. For a more thorough account of these issues and their relationship to program development cf. e.g. [18].

6 The case study

We are going to study the modular implementation in Lego of the symbol table example which is investigated in [24]. We chose this example both because it has many features capable of demonstrating the power of a specification language and because it allows to compare our approach with other work. The version desribed here is a revised and shortened account of the author's MSc thesis [14].

We do not describe all the proofs in full detail here. The complete Lego files together with a description of their usage can, however, be obtained from the author by ftp (see Appendix) or on e-mail request.

6.1 Initial specification

Our task is to implement a symbol table — a data structure capable of storing information about *identifiers* (this information is called *attributes*); a pair of an identifier and its attribute forms an *entry*. The functions operating on the symbol table should allow to add new entries (`add`), to modify existing entries (`change_attrib`) and to extract the attribute corresponding to an identifier (`lookup`). Moreover we expect a decision procedure for the presence of an identifier in a symbol table and finally a value corresponding to the empty symbol table. The implementation should be parametrised on the specific choice of types of identifiers (`IDENT`) and attributes (`ATTRIB`). Our ultimate goal will then be:

```
Goal {Ident:IDENT}{Attrib:ATTRIB} SYMTAB Ident Attrib;
```

where the signatures `IDENT`, `ATTRIB` and `SYMTAB` are as follows.

```
[IDENT =
          <ident :  Prop>
          <eqid :  ident -> ident -> bool>
          (eqtype ident eqid)
];
```

Here `eqtype` is a defined predicate stating that `eqid` is indeed the equality function on `ident`:

```
[eqtype [X:Prop][eq:X->X->bool] =
         {x,y:X}iff (isthecase (eq x y)) (Q x y)];
```

where `[isthecase[b:bool] = Q true b]` is the obvious conversion function from `bool` to `Prop`. Hence an inhabitant of `IDENT` consists of a type and a (provably correct) equality function on it. More precisely, if `Ident` is a structure of type `IDENT` then `Ident.2.1` is an equality function on `Ident.1` (the underlying type) and `Ident.2.2` proves its correctness. Since we are using intuitionistic logic without the principle of the excluded middle, we make a distinction between the decidable equality on identifiers and attributes and the propositional equality which is defined for all types, but lacks a (boolean–valued) characteristic function.

The signature of attributes is given by

```
[ATTRIB =
         <attrib :  Prop>
         <eqattr :  attrib -> attrib -> bool>
         <null_attrib :  attrib>
         (eqtype attrib eqattr)
];
```

Here again an equality type[9] is specified; additionally we require a default value (`null_attrib`) which will be returned by the `lookup` function if the required identifier happens to be absent. (In future we would like to model this using exceptions).

Remark: It would also be possible to define a Σ-type `eqtype` which incorporates both a type and a provably correct equality function on it in such a way that `ATTRIB` would become:

```
[ATTRIB =
         <attrib :  eqtype>
         <null_attrib :  attrib.1>
];
```

6.1.1 The signature SYMTAB

Since later the `IDENT` and `ATTRIB` components of this signature (figure 3) will appear in sharing constraints, it depends upon arbitrary implementations of `IDENT` and `ATTRIB`

```
[SYMTAB [Ident :  IDENT] [Attrib :  ATTRIB] = ...
```

The explicit typing of the unpacked parts as in

```
[eqid :  ident -> ident -> bool = Ident.2.1]
```

[9]The need for an equality function here is explained in 6.3.2

```
[SYMTAB [Ident :  IDENT] [Attrib :  ATTRIB] =

                Unpacking the argument structures
    [ident = Ident.1]
    [eqid :  ident -> ident -> bool = Ident.2.1]
    [EQID : eqtype ident eqid = Ident.2.2]
    [attrib = Attrib.1]
    [eqattr :  attrib -> attrib -> bool = Attrib.2.1]
    [null_attrib :  attrib = Attrib.2.2.1]
    [EQATTR : eqtype attrib eqattr = Attrib.2.2.2]

                Type and Functions
    <symtab :  Prop>
    <empty :  symtab>
    <add :  ident -> attrib -> symtab -> symtab>
    <change_attrib :  ident -> attrib -> symtab -> symtab>
    <present :  ident -> symtab -> bool>

                Axioms
      ({i :  ident} isnotthecase (present i empty)) #


      ({i,i' :  ident}{a' :  attrib}{s :  symtab}
                Q (present i (add i' a' s))
                   (orelse (eqid i i') (present i s))) #

      ({i,i' :  ident}{a' :  attrib}{s :  symtab}
                Q (present i (change_attrib i' a' s))
                   (present i s)) #

                The lookup function
    <lookup :  ident -> symtab -> attrib>

                Axioms for lookup
      ({i :  ident}Q (lookup i empty) null_attrib) #

      ({i :  ident}{a :  attrib}{s :  symtab}
                Q (lookup i (add i a s)) a) #

      ({i,i' :  ident}{a' :  attrib}{s :  symtab}
                (isnotthecase (eqid i i')) ->
                   (Q (lookup i (add i' a' s)) (lookup i s))) #

      ({i :  ident}{a :  attrib}{s :  symtab}
                (isthecase (present i s)) ->
                   (Q (lookup i (change_attrib i a s)) a)) #

      ({i,i' :  ident}{a' :  attrib}{s :  symtab}
                (isnotthecase (eqid i i')) ->
                   (Q (lookup i (change_attrib i' a' s)) (lookup i s)))
];
```

Figure 3: The signature SYMTAB

is convenient since otherwise `eqid` would have type `Ident.1->Ident.1->bool` which is equivalent but less readable. For the future it might be worthwhile to extend Lego by Σ–types with named projections which would handle these issues automatically.

The function `isnotthecase` states falsity of a boolean expression and is defined by `[isnotthecase[b : bool] = Q false b]`

In order to stay close to the original example in [24] we have used first–order equational logic throughout. As an example we show how we could specify `present` in a PROLOG–style way:

```
...
{i : ident}{s : symtab}
iff (isthecase (present i s))
     ({Present : ident -> symtab -> Prop}
         ({a : attrib}{s : symtab}
             Present i (add i a s)) ->                              (I)
         ({i' : ident}{a : attrib}{s : symtab}
             (Present i s) -> (Present i (add i' a s))) ->          (II)
     Present i s)
...
```

The two clauses marked (I) and (II) above form the actual inductive definition. The rest is a canonical encoding of inductive predicates into higher–order logic. This method avoids the need for the first negative clause and is — we believe — also more intuitive.

It is left as a (not straightforward) exercise to prove in Lego that a function which meets this specification satisfies the equations given in the actual specification. (The other direction is true only under the assumption that all symbol tables are generated by `add` and `change_attrib`).

6.2 First decomposition step

Notice that if by some magic we succeed in solving the original goal from page 14 we will have constructed both a program implementing `SYMTAB` and a proof of its correctness. Of course it is too complicated to achieve this in one step; we therefore solve the task by means of several refinement steps. The first of these steps consists of implementing the `change_attrib` function by just adding the modified entry using `add`. We therefore assume that everything but `change_attrib` is already implemented and solve our original task modulo this assumption. To that end we first define a signature `SYMTAB'` which is like `SYMTAB` but lacks the entries for `change_attrib` (i.e. the function itself and the three axioms mentioning it) and then try to implement a functor of the type:

```
[CHANGE_ATTRIB =
{Ident : IDENT}{Attrib : ATTRIB}
      (SYMTAB' Ident Attrib) -> SYMTAB Ident Attrib];
```

We may check that our decomposition typechecks by assuming functors:

```
[Change_Attrib : CHANGE_ATTRIB]
[Symtab' : {Ident : IDENT}{Attrib : ATTRIB}SYMTAB' Ident Attrib]
```

and composing them:

```
[Symtab[Ident : IDENT][Attrib : ATTRIB] : SYMTAB Ident Attrib =
   Change_Attrib Ident Attrib (Symtab' Ident Attrib)];
Forget Change_Attrib;
```

If we had made an error in the definition of `CHANGE_ATTRIB` or the composition of the two hypothetical functors, the Lego typechecker would have shown it. The `Forget` command at the end removes the hypothetical functors (not only `Change_Attrib`, but also `Symtab` and `Symtab'`) since we are going to actually implement them. Our first task consists of implementing `Change_Attrib`. We may either type in:

```
Goal CHANGE_ATTRIB;
```

and using the `Refine` mechanism to solve the goal interactively, or rather build up the functor by subsequent definitions as follows. We assume the arguments of the functor ...

```
[Ident : IDENT][Attrib : ATTRIB]
[St' : SYMTAB' Ident Attrib];
```

... and "unpack" them such that all components become accessible by their names:

```
[ident = Ident.1]
[eqid:ident->ident->bool = Ident.2.1]
...
[symtab = St'.1]
[empty:symtab = St'.2.1]
[add:ident->attrib->symtab->symtab = St'.2.2.1]
[present:ident->symtab->bool = St'.2.2.2.1]
[presempty:{i:ident}isnotthecase (present i empty) = St'.2.2.2.2.1]
```

and similarly for the other components of `Ident`, `Attrib`, and `St'`: `EQID`, `attrib`, `null_attrib`, `eqattr`, `EQATTR`, `presadd`, `lookup`, `lkupempty`, `lkupadd1`, `lkupadd2`. Notice that `presempty`, `presadd`, etc. are hypothetical proof terms for the corresponding axioms which we will subsequently use as "lemmas".

Now we can implement `change_attrib` using a case distinction:

```
[change_attrib[i : ident][a : attrib][s : symtab] =
    if (present i s)
        (add i a s)                                        then-part
        s];                                                else-part
```

In traditional program development this is already the whole job, whereas in Lego we must formally construct a correctness proof of this implementation, more precisely we must prove the three axioms concerning `change_attrib`. We start with the second axiom stating that after changing an attribute the new information becomes available by `lookup`. To give an impression of the actual interaction with Lego, we give the full transcript here. The text preceded by question marks shows the actual goals; this (and all the other indented text) is output by the system:

```
Goal {i : ident}{a : attrib}{s : symtab}
          (isthecase (present i s)) ->
          (Q (lookup i (change_attrib i a s)) a);
Intros i a s isp;                    We assume i, a, s and that the hypothesis holds
   i : ident
   a : attrib
   s : symtab
   isp : isthecase (present i s)
   ?1 : Q (lookup i (change_attrib i a s)) a
Refine Q_trans;                                          We use transitivity of Q ...
Refine (lookup i (add i a s));                     ...with this as intermediate value
   ?6 : Q (lookup i (change_attrib i a s)) (lookup i (add i a s))
   ?7 : Q (lookup i (add i a s)) a
Refine Q_resp;                               Q respects the application of lookup i
   ?13 : Q (change_attrib i a s) (add i a s)
   ?7 : Q (lookup i (add i a s)) a
Refine Q_sym;                                                 transposes the equation
   ?17 : Q (add i a s) (change_attrib i a s)
   ?7 : Q (lookup i (add i a s)) a
Refine iftrue;                                                              see below
   ?22 : isthecase (present i s)                     this is one of the hypotheses
   ?7 : Q (lookup i (add i a s)) a                 this is an instance of an axiom
Refine isp;                                      using the hypothesis to solve ?22
Refine lkupadd1;                                using the axiom from St' to solve ?7
   *** QED ***                                                the goal is proven
Save chlkup1;                           we save its proof term under the name chlkup1
```

Here we have used a lemma[10] called `iftrue` which states

```
{A : Prop}{a1,a2 : A}{b : bool}(isthecase b) -> Q a1 (if b a1 a2)
```

Similarly there is a lemma `iffalse` stating:

```
{A : Prop}{a1,a2 : A}{b : bool}(isnotthecase b) -> Q a2 (if b a1 a2)
```

We omit the (trivial) proofs of both. Next we prove the axiom describing the relationship between `present` and `change_attrib` (it is the first of the three axioms mentioning `change_attrib` in the signature `SYMTAB`).

```
Goal {i,i' : ident}{a' : attrib}{s : symtab}
                Q (present i (change_attrib i' a' s))
                  (present i s);
Intros i i' a' s;                                          introducing the parameters
    i : ident
    i' : ident
    a' : attrib
    s : symtab
    ?1 : Q (present i (change_attrib i' a' s)) (present i s)
Expand change_attrib;                       expanding the definition of change_attrib
?1 : Q (present i (if (present i' s) (add i' a' s) s)) (present i s)
```

[10] By lack of a better expression we use here and leater the word "lemma" for mathematically trivial, yet very useful propositions describing the definitions. Is some sense they correspond more to tactics rather then to actual mathematical statement, cf. for example the lemma "bycases" which appears on the next page

Here an `if`–expression occurs inside a context; using the lemma `ifind`, which we are going to describe, we can handle the `then` and the `else` parts separately. This lemma — a kind of induction principle for `if`–expressions — states:

```
{A : Prop}{a1,a2 :  A}{P : A -> Prop}{b :  bool}
    ((isthecase b) -> P a1) ->
    ((isnotthecase b) -> P a2) ->
       P (if b a1 a2)
```

I.e. to prove an arbitrary property (`P`) of an `if`-expression we must show it for the `then`-part under the assumption that the boolean value equals `true` and for the `else` part under the opposite assumption. In our situation the property `P` (i.e. the context in which the expression occurs) is:

```
[P [x :  symtab] = Q (present i x) (present i s)];
```

The arguments `A`, `a1` and `a2` of `ifind` need not be given explicitly — they can be replaced by question marks and will be inferred by the system.

```
Refine ifind ?  ?  ?  ([x:symtab]Q (present i x)(present i s))
    ?7 :  (isthecase (present i' s)) -> Q (present i (add i' a' s)) (present i s)
    ?8 :  (isnotthecase (present i' s)) -> Q (present i s) (present i s)
```

Observe that these two subgoals reflect quite closely the line of an intuitive verification of `change_attrib` ("Suppose `i'` is present in `s`; then `change_attrib` equals `add ...`"). By introducing the hypothesis `(isthecase (present i' s))` and using the equation describing the relationship between `present` and `add` we arrive at the situation

```
i :  ident
i' :  ident
a' :  attrib
s :  symtab
isp :  isthecase (present i' s)
    ?15 :  Q (orelse (eqid i i') (present i s)) (present i s)
```

Here we must make a case distinction upon whether `i` equals `i'` or not. This is possible since `ident` comes with a boolean–valued equality function. The fact that boolean values satisfy the law of the excluded middle is stated in the lemma `bycases`:

```
{b :  bool} or (isthecase b) (isnotthecase b)
```

whose name is influenced by its actual usage:

```
Refine bycases (eqid i i');
    ?21 :  (isthecase (eqid i i')) ->
            Q (orelse (eqid i i') (present i s)) (present i s)
    ?22 :  (isnotthecase (eqid i i')) ->
            Q (orelse (eqid i i') (present i s)) (present i s)
```

The rest of the proof consists essentially of equality manipulation (we must use `EQID` which proves that `eqid` reflects Leibniz equality) and some calculations with `orelse`.

```
    *** QED ***
Save chpres;
```

The axiom `chlkup2` which states that `change_attrib` does not affect other entries follows almost immediately from the axiom `lkupadd2` which states a similar property for `add`. Now we can put all the terms together to obtain an instance of `Symtab`:

```
[ChangeAttrib = (
symtab,
empty, add, change_attrib, present,
presempty, presadd, chpres,
lookup,
lkupempty, lkupadd1, lkupadd2, chlkup1, chlkup2) :  SYMTAB Ident Attrib];
```

By the command

```
Discharge Ident;
```

we turn `Change_Attrib` into a function depending on the previously assumed arguments `Ident`, `Attrib` and `St'`. We may check this by printing `Change_Attrib` together with its type:

```
ChangeAttrib;
    value = ...
    type = {Ident :  IDENT}{Attrib :  ATTRIB}(SYMTAB' Ident Attrib) ->
        SYMTAB Ident Attrib
```

which indeed turns out to be the expected one.

6.3 Second decomposition step

The next step consists of implementing the signature `SYMTAB'` in order to provide an argument to the functor just implemented. We must make a design decision how to realise the type `symtab` which stores the entries. From several possibilities (see [24]) we choose the implementation by duplicate–free lists of ⟨identifier, attribute⟩–pairs. To that end we introduce a signature `SEQDUP` describing a list structure together with functions dealing with the removal of duplicates. The signature `SEQDUP` in turn will depend upon a signature `SEQPAIRS` which provides a basic list structure without the removal facilities. Our goal will then be to construct the following functors

```
Symtab'  :{Ident :  IDENT} {Attrib :  ATTRIB}
          {Seq :  SEQPAIRS Ident Attrib}
          (SEQDUP Ident Attrib Seq) -> (SYMTAB' Ident Attrib)

SeqDup   :{Ident :  IDENT} {Attrib :  ATTRIB}
          {Seq :  SEQPAIRS Ident Attrib} (SEQDUP Ident Attrib Seq)

SeqPairs :{Ident :  IDENT} {Attrib :  ATTRIB} (SEQPAIRS Ident Attrib)
```

which, given `Ident` and `Attrib` can then be plugged together with the previously constructed functor `ChangeAttrib` in the following way

```
[Seq = SeqPairs Ident Attrib];

[Symtab :  SYMTAB Ident Attrib =
    ChangeAttrib Ident Attrib
       (Symtab' Ident Attrib Seq
          (SeqDup Ident Attrib Seq))];
```

to obtain the required `SYMTAB` structure. The rest of this paper is mainly devoted to the construction of the functor `Symtab'` since the two others do not contain new interesting features. The construction of the functor `Symtab'`, however, demonstrates the following new aspects:

- Use of structural induction for lists in Lego
- Use of local definitions in a specification
- An example of "general recursion" as described in section 3 and the corresponding proof principle
- The use of a higher–order definition to express that the elements of some type are generated only by certain functions

We will only briefly describe these features here; a longer account can be found in [14].

6.3.1 The signature `SEQPAIRS`

This signature (figure 4) provides a type `sequence` of lists of symbol table entries. The functions `nil` and `cons` together with the recursor `seqrec` and the induction principle correspond to an SML datatype declaration, i.e. they enforce that `sequence` will be realised by an initial algebra. The other entries of the signature add an emptiness test (`null`) and inverses to the constructors (`hd` and `tl`). Notice here that the behaviour of the latter on the empty list remains unspecified. This implies that the correctness proofs for functions using `hd` and `tl` will not depend upon the specific default value chosen.

6.3.2 The signature `SEQDUP`

This signature (figure 5) enriches `SEQPAIRS` by three new features. The function `ismatch` checks whether there exists an entry for some identifier and will later (in `Symtab'`) implement the `present` function. The function `add` puts a new entry into a sequence thereby ensuring the presence of at most one entry for each identifier using the function `remove`. The specification of `remove` involves a locally defined membership test. To understand the actual content of this specification we will rephrase the last axiom concerning `remove` in Extended ML notation:

```
[SEQPAIRS [Ident : IDENT] [Attrib : ATTRIB] =

[ident=Ident.1] [attrib=Attrib.1]


               Type and Functions

<sequence : Prop>
<nil : sequence>
<cons : (PROD ident attrib) -> sequence -> sequence>


               Recursion and Induction Principles

<seqrec : {R | Prop} R -> ((PROD ident attrib) -> sequence -> R -> R)
                                      -> sequence -> R>
({P : sequence -> Prop}(P nil) ->
      ({e : PROD ident attrib}{s : sequence}
               (P s) -> P (cons e s)) ->
      {s : sequence}P s) #

({R | Prop}{g : R}{h : (PROD ident attrib) -> sequence -> R -> R}
    Q g (seqrec g h nil)) #
({R | Prop}{g : R}{h : (PROD ident attrib) -> sequence -> R -> R}
 {e : PROD ident attrib}{s : sequence}
    Q (h e s (seqrec g h s))(seqrec g h (cons e s))) #


               Auxiliary Functions with Axioms

<null : sequence -> bool>
((isthecase (null nil)) #
({e : PROD ident attrib}{s : sequence}isnotthecase (null (cons e s))) #

<hd : sequence -> PROD ident attrib>
({e : PROD ident attrib}{s : sequence} Q (hd (cons e s)) e) #

<tl : sequence -> sequence>
({e : PROD ident attrib}{s : sequence} Q (tl (cons e s)) s)];
```

Figure 4: The signature SEQPAIRS

```
[SEQDUP [Ident :  IDENT] [Attrib :  ATTRIB] [Seq :  SEQPAIRS Ident Attrib] =
                Unpacking the argument structures

[ident = Ident.1][eqid :  ident -> ident -> bool = Ident.2.1]
[attrib = Attrib.1] [sequence = Seq.1][nil = Seq.2.1][cons = Seq.2.2.1]

                Functions
<add :  (PROD ident attrib) -> sequence -> sequence>
<ismatch :  ident -> sequence -> bool>
<remove :  ident -> sequence -> sequence>

                Axioms for add
({i :  ident}{a :  attrib}{s :  sequence} (isthecase (ismatch i s)) ->
    Q (add (pair i a) s)
      (cons (pair i a) (remove i s))) #

({i :  ident}{a :  attrib}{s :  sequence}(isnotthecase (ismatch i s)) ->
    Q (add (pair i a) s)
      (cons (pair i a) s)) #

                Axioms for ismatch
({i :  ident} isnotthecase (ismatch i nil)) #

({i :  ident}{i' :  ident}{a' :  attrib}{s :  sequence}
    Q (ismatch i (cons (pair i' a') s))
      (orelse (eqid i i') (ismatch i s))) #

                Axioms for remove involving a local function
({i :  ident}{s :  sequence}isnotthecase (ismatch i (remove i s))) #

local definitions
    (Ex [member:(PROD ident attrib)->sequence->bool]
    Ex [eqe:(PROD ident attrib)->(PROD ident attrib)->bool]
    Ex [eqt:eqtype (PROD ident attrib) eqe]

                Local Axioms for member
    Ex [memnil:{e:PROD ident attrib}isnotthecase (member e nil)]
    Ex [memcons:{e,e':PROD ident attrib}{s:sequence}
    Q (member e (cons e' s)) (orelse (eqe e e')(member e s))]
used in
{i,i' :  ident}{a' :  attrib}{s :  sequence}(isnotthecase (eqid i i')) ->
    Q (member (pair i' a') (remove i s))
      (member (pair i' a') s)
)];
```

Figure 5: The signature SEQDUP

```
local
    val member :  ident * attrib -> sequence -> bool
    axiom (member e nil) = false
    axiom (member e (cons e' s)) = (e=e') orelse (member e s)
in
    axiom i<>i' => (member (i',a') (remove i s)) = (member (i',a') s)
end
```

There are in principle two possibilities to understand local functions type theoretically. The first one which we use here defers the "implementation" of the local function to the implementation of the signature in which it occurs. When using the signature hypothetically (as usual in top down development) one may assume the existence of a function satisfying the local axioms. The `local` construct is therefore modeled by a weak existential quantification (`Ex`). Its weakness disallows the use of the local function in the actual implementation. It may only be used to *prove* propositions involving functions whose specification depends on the local function. This reflects the intended information hiding.[11]

To make this more precise let us have a look at the actual definition (in figure 2) of the existential quantifier. If `A` is a type (in our case the type of the `member` function) and `P : A -> Prop` is a predicate on `A` then `Ex[a:A]P a` expands to

```
{B : Prop} ({a :  A} (P t) -> B ) -> B
```

This means that in the presence of a proof of `Ex[a:A]P a` to prove an arbitrary proposition `B` it is enough to prove it under the assumption of a fresh variable `a` of type `A` and a proof that it has the property `P`. This means in our example that to prove a proposition involving `remove` we may assume a hypothetical function `member` which meets its specification (which again contains a local function, namely `eqe`).

To prove the existential statement (in our case this happens during the implementation of the `SEQDUP` signature) one must supply a witness and a proof that it satisfies the required property. This witness and its correctness proof, however, cannot be extracted from the proof of the existential statement *a posteriori*. This means that although we may assume the `member` function to prove properties of `remove` we cannot use it in the actual implementation. This reflects the intended information hiding, as already mentioned. Moreover if we use — which we shall not do in this work — classical logic at the propositional level, we may use local functions which cannot really be implemented. (For example using classical logic we can prove that there exists a function which decides the halting problem.)[12]

Another way of modelling local functions consists of using universal quantification; in this case we have to supply a function (here `member`) at each usage of the axiom involving it. This, however, does not seem adequate.

[11] In our presentation where we keep propositions and types in the same universe this is not exactly true since in the presence of a strong sum — which we shall use later on to model `symtab` as a subtype — the weak existential quantifier becomes strong. This is one more reason why we would like to separate propositions and types in forthcoming versions.

[12] Thanks to E. Kazmierczak for this comment.

Note that in either case we must prove the existence of the local function at some stage — either at the point odf use or at the implementation of the involved structure. This is the reason why we need a decidable equality on attributes (see section 6.1). In [10] it is suggested, that what one actually has to do is to supply a proof of *persistency* of the local specification over the context in which it occurs. Usually this is in fact done by showing that there exists an implementation, but there may be some cases in which other methods are conceivable.

6.3.3 Implementation of `Symtab'`

We will now show how to obtain an implementation of `SYMTAB'` (this is the original specification without the `change_attrib` facility) from an implementation of `SEQDUP`. To that end we assume a structure implementing `SEQDUP` and "unpack" it as before thereby introducing names for the (proofs of the) axioms:

```
            The arguments to SeqDup

[Ident :  IDENT][Attrib :  ATTRIB]]
[Seq :  SEQPAIRS Ident Attrib ]

            Opening the arguments
[ident = ...]                                   opening IDENT and ATTRIB as before
[Ssequence = Seq.1]
[Snil :  Ssequence = Seq.2.1] [Scons = ...]          the type with its constructors
[Sseqind = ...]                                          the induction principle
[Sseqrec = ...]                                          the recursion principle
[Snilrec = ...]  [Sconsrec = ...]                       equations for the recursor
[Snull = ...][Snull1:  ...]  [Snull2 = ...]          emptiness test with equations
[Shd = ...]  [Shdax = ...]
[Stl = ...  ][Stlax = ...]                      hd and tl with specifying equations


            The actual SEQDUP structure

[S:SEQDUP Ident Attrib Seq]

[Sadd = S.1 ][Sismatch = S.2.1][Sremove = ...]
[Sadd1 = ...]  [Sadd2 = ...]                                     equations for add
[Sismatch1 = ...][Sismatch2 = ...]                           equations for ismatch
[Sremove1 = ...]                                              first axiom for remove
[Sremove2 = ...]                     second axiom involving the local specification
```

The "S" prefixes were introduced to avoid name clashes with functions to be defined later. Starting from these data we must construct a (verified) implementation of the signature `SYMTAB'`. Recall that this is exactly the same as `SYMTAB` (figure 3) except that it lacks `change_attrib` and the corresponding axioms. As before we shall construct the ingredients step by step and put them together afterwards. We start by defining the type `symtab`.

6.3.3.1 Implementation of `symtab`

One would be tempted to put `[symtab = Ssequence]`. Then, however, we may not assume in the proofs that each element of `symtab` contains at most one entry. As it happens we need that assumption to prove the `lookup` function correct. This is because we have no guarantee that `remove` does not affect the order of entries (it is specified only that inappropriate entries are neither deleted nor added). Of course, if we generate a symbol table just by using the functions `Snil` and `Sadd`, we may assume uniqueness of entries. To make this assumption available for the proofs we must "decorate" the elements of `symtab` with proofs that they were generated using only these two functions. To that end we define a predicate on `Ssequence` expressing this:

```
[gen_by_add[s : Ssequence] =
{P : Ssequence -> Prop} (P Snil) ->
    ({s : Ssequence}{e : PROD ident attrib }
       (P s) -> P (Sadd e s) ) ->
    P s]
```

If for some `s:Ssequence` the statement `(gen_by_add s)` holds then each property which holds for `Snil` and is preserved under `Sadd` holds for `s`, too. Using this predicate we shall define `symtab` as a Σ–type:

```
[symtab = <s : Ssequence> gen_by_add s]
```

An element `x` of `symtab` now has two components. The first, `x.1`, is an element of `Ssequence`; the second, `x.2`, is a proof of `(gen_by_add x.1)`. Now in order to define the functions `empty` and `add` for `symtab` we must "transport" these proofs:

```
[empty =
(
   Snil,                                                    the value
   [P : Ssequence -> Prop][Pnil : P Snil]
      [_:{s:Ssequence}{e:PROD ident attrib}(P s) -> P(Sadd e s)]
        Pnil                                 proof of (gen_by_add Snil)
) : symtab]
```

The definition of `add` is only slightly more complicated, here we must use the fact that the argument is already a `symtab` hence comes with a proof that it is generated from `Snil` by `Sadd`. Moreover we "curry" the `Sadd` function to match it with the signature `SYMTAB'`:

```
[add[i:ident][a:attrib][s:symtab] =
(
   Sadd (pair i a) s.1,                                     the value
   [P:Ssequence->Prop][Pnil:P Snil]
      [Pstab:{s:Ssequence}{e:PROD ident attrib}(P s)->P(Sadd e s)]
        Pstab s.1 (pair i a) (s.2 P Pnil Pstab)             the proof term
):symtab];
```

Remark: Notice that since we have solved the problem by mere definitions without assuming any further axioms, we can guarantee the soundness of our solution.

6.3.3.2 Implementation and verification of `present` and `add`

The implementation of `present` is immediate from `Sismatch`; we only have to forget the proof component:

```
[present :  ident -> symtab -> bool =
    [i :  ident][s :  symtab] Sismatch i s.1];
```

Before implementing the last function (`lookup`), we will carry out the verification of the functions defined so far. We must show:

```
Goal {i:ident}isnotthecase (present i empty);
```

and

```
Goal {i,i':ident}{a':attrib}{s:symtab}
                 Q (present i (add i' a' s))
                   (orelse (eqid i i')(present i s));
```

The first goal following immediately from `Sismatch1`; we only discuss the axiom describing `add`. It follows by simple equality manipulations and case distinctions if we manage to prove the following lemma which is also used in the original work [24]:

```
Goal {i,i' :  ident}{s :  Ssequence}
(isnotthecase (eqid i i')) ->
     Q (Sismatch i (Sremove i' s)) (Sismatch i s);
```

The proof of this lemma is a bit more complicated since the properties of `Sremove` are known only in the context of the locally–defined `member` function. So our first step consists of establishing a connection between `member` and `Sismatch`, e.g.

```
(isthecase (Sismatch i s))->
Ex ([a:attrib]isthecase (member (pair i a) s));
```

Unfortunately, however, the `member` function is not accessible outside the local specification of `Sremove`. So we must universally quantify the statement by an arbitrary function satisfying the axioms for `member`:

```
Goal {i:ident}{s:Ssequence}
     {member:(PROD ident attrib)->Ssequence->bool}
     {eqe:(PROD ident attrib)->(PROD ident attrib)->bool}
     {eqt:eqtype (PROD ident attrib) eqe}
     {memnil:{e:PROD ident attrib}isnotthecase (member e Snil)}
     {memcons:{e,e':PROD ident attrib}{s:Ssequence}
            Q (member e (Scons e' s)) (orelse (eqe e e')(member e s))}
     (isthecase (Sismatch i s)) ->
            Ex ([a:attrib]isthecase (member (pair i a) s))
```

Since the proof of this lemma uses list induction we show the beginning of it here. By the command

```
Intros i s member eqe eqt memnil memcons;
```

we arrive at the more familiar situation

```
    i :  ident
    s :  Ssequence
    member :  (PROD ident attrib) -> Ssequence -> bool
    eqe :  (PROD ident attrib) -> (PROD ident attrib) -> bool
    eqt :  eqtype (PROD ident attrib) eqe
    memnil :  {e :  PROD ident attrib} isnotthecase (member e Snil)
    memcons :  {e,e' :  PROD ident attrib}{s :  Ssequence}
                Q (member e (Scons e' s)) (orelse (eqe e e') (member e s))
    ?1 :  (isthecase (Sismatch i s)) ->
                Ex ([a :  attrib] isthecase (member (pair i a) s))
```

We solve this goal by induction (`Sseqind`) along `s`. To that end we must instantiate `Sseqind` by the actual predicate we want to show for `s`.

```
[P[s :  Ssequence] = (isthecase (Sismatch i s)) ->
        Ex ([a :  attrib]isthecase (member (pair i a) s))];

Refine Sseqind P;
    ?2 :  P Snil                                                      base case
    ?3 :  {e:PROD ident attrib}{s:Ssequence} (P s) -> (P (Scons e) s)   ind. case
```

The base case now follows from a contradiction of its hypothesis (`isthecase (Sismatch i Snil)`) with the axiom `Sismatch1`. For the inductive case we must make a case distinction and then use either the "consed" identifier or the inductive hypothesis to supply a witness for the existential statement. We only show the second case. The situation looks as follows:

```
    e :  PROD ident attrib
    s1 :  Ssequence
    indhyp :  (isthecase (Sismatch i s1))->
                 Ex ([a:attrib]isthecase (member (pair i a) s1))
    H : isthecase (Sismatch i (Scons (pair (pi1 e) (pi2 e)) s1))
    ism :  isthecase (Sismatch i s1)
    ?91 :  Ex ([a:attrib]isthecase
                 (member (pair i a) (Scons (pair (pi1 e) (pi2 e)) s1)))
```

Now we open the induction hypothesis by

```
Refine indhyp ism;
Intros a;Intros ismem;
    a :  attrib
    ismem :  isthecase (member (pair i a) s1)
    ?95 :  Ex ([a:attrib]isthecase (member (pair i a)
                        (Scons (pair (pi1 e) (pi2 e)) s1)))
Refine ExIntro;Refine a;                                 using a as a witness
    ?99 :  isthecase (member (pair i a) (Scons (pair (pi1 e) (pi2 e)) s1))
```

That a satisfies the requirement and hence the whole lemma then follows from ismem and the second axiom on member.

```
Save ismtomem;
```

It may seem strange that even in an intuitionistic framework we succeeded in effectively giving (Ex[a:attrib]) the corresponding attribute from the mere assertion of the presence of an entry (isthecase (Sismatch ...)). The reason is — as we have seen — that during the induction all entries of the list become accessible. The computational content of our proof is a primitive recursive algorithm which step by step unravels the list.

To carry out the proof of the lemma on page 28 we also need the other direction of ismtomem and the contrapositives of both. The lemma itself then follows easily.

6.3.3.3 Implementation of the lookup function

We will use the following recursive definition:

```
fun lookup i s = if (Snull s) then null_attrib
                else let val (i',a') = Shd s in
                        if i=i' then a'
                        else (lookup i (Stl s));
```

We translate this into Lego using the definition "recurse" described in section 3. Since it requires the recursive variable to be in the first position and also because the proof component of symtab is not actually needed here, we are going to define by recursion a function lookup_tw of type

```
Ssequence -> ident -> attrib
```

from which we derive lookup by

```
[lookup:ident->symtab->attrib =
    [i:ident][s:symtab]lookup_tw s.1 i]
```

For the recursive definition of lookup_tw we must supply a suitable complexity measure. Here this is of course the length function which we define by primitive recursion:

```
[Slength:Ssequence->nat = Sseqrec
    zero
    ([_:PROD ident attrib][_:Ssequence][n:nat]succ n)];
```

The next step is the definition of the base constant, i.e. the value lookup_tw takes on if the termination condition (here Snull) is satisfied:

```
[lkup_base=[_:Ssequence][_:ident]null_attrib]
```

The actual recursion scheme is given by

```
[lkup_scheme[s:Ssequence][lk:Ssequence->ident->attrib] =
  [i:ident] let (Shd s) ([e:PROD ident attrib][i'=pi1 e][a'=pi2 e]
            if (eqid i i') a'
                           (lk (Stl s) i))]
```

where let is defined by

```
[let[A,B|Prop] = [a:A][f:A->B]f a];
```

Now we can implement lookup_tw by putting

```
[lookup_tw:Ssequence->ident->attrib =
   recurse Snull lkup_base lkup_scheme Slength]
```

In order to reason about this definition we must show that the complexity measure is actually reduced by the recursion scheme and that it implies the termination condition Snull if it reaches zero. This is done by the following two goals:

```
Goal {s:Ssequence}(Q (Slength s) zero)->isthecase (Snull s);
...
Save lkup_bc;

Goal {s:Ssequence}or (isthecase (Snull s))
       ({f1,f2:Ssequence->ident->attrib}({s':Ssequence}
          (LT (Slength s') (Slength s))->
               Q (f1 s') (f2 s'))->
   Q (lkup_scheme s f1) (lkup_scheme s f2));
...
Save lkup_cc;
```

The functions LT and LE used here and in the sequel have type nat->nat->Prop, instead of nat->nat->bool which is the type of the function le used in Section 3. Notice here that the fact that lookup_scheme reduces the length is expressed by the fact that (lkup_scheme s) produces an equal reaction on two functions which behave the same for sequences shorter than s. The proof of these two goals is by induction. We shall use these proofs to instantiate two lemmas (recurse_law and rec_ind) which are proved for arbitrary recursion scheme (for the proofs see the appendix). These lemmas establish that lookup_tw satisfies its recursive definition and an induction principle similar to Scott induction the use of which we shall demonstrate in the next step.

6.3.3.4 Proof of a lemma concerning lookup

We want to show that lookup_tw actually returns the default value null_attrib in case that the requested identifier is not present:

```
Goal {i:ident}{s:Ssequence}
   (isnotthecase (Sismatch i s))->Q null_attrib (lookup_tw s i);
```

This will be needed in the proof of the third axiom for lookup which we do not describe here. The proof principle rec_ind describing recursively defined functions applies to goals of the form

```
 ?1 :  {a:A}P a (recurse some recursion scheme)
```

where

```
[P:A -> (A->B) -> Prop]
```

Notice that P involves the recursively defined function itself and not just some specific values of it. We must then show that

- the predicate P evaluates its functional argument only up to some value depending on the first argument
- that P holds in the base case
- that P is propagated along the recursion scheme

In our example this looks as follows:

```
Intros i;                                              This applies to the goal above
[P[s:Ssequence][f:Ssequence->ident->atrib] =
       (isnotthecase (Sismatch i s))->Q null_attrib (f s i)];

Refine rec_ind ?  ?  ?  ?  lkup_bc lkup_cc P Slength;Intros _H;Refine H;
```

Here lkup_bc and lkup_cc are the previously constructed proofs for the termination of the lookup recursion.

```
  ?9 :  {s:Ssequence}{f1,f2:Ssequence->ident->attrib}
     ({s':Ssequence}(LE (Slength s') (Slength s)) -> (Q (f1 s') (f2 s'))) ->
        (P s' f1) -> (P s' f2)

  ?10 :  {s:Ssequence}(Q (Slength s) zero)->
        P s ([_:Ssequence]lkup_base)

  ?11 :  {f:Ssequence->ident->attrib}{s:Ssequence}
        ({s':Ssequence}(LT (Slength s') (Slength s))->P s' f)->
        P s ([s:Ssequence]if (Snull s) lkup_base (lkup_scheme s f))
```

The subgoal `?9` expresses that in `P s f` the second argument is evaluated only at sequences whose length is less or equal to the one of `s`. This is immediate from the definition of `P`. In `?10` we must prove `P` for the base case under the additional assumption that the length of `s` is zero. This assumption is not needed here since `lkup_base i` equals `null_attrib` for all `i`. The last goal `?11` states the stability of `P` under the recursive scheme. Its proof is messy but straightforward.

It should be remarked that the `rec_ind` principle has itself been derived in Lego from the definiton of `recurse` and hence is provably correct. Of course in this case also a careful examination by hand calculation would yield an adequate proof principle, but in more difficult cases the machine checking also of the derivation of inference rules becomes very useful. We see here an advantage of our approach compared to the use of the machine as a mere "logical framework" which processes arbitrary user–declared proof rules.

The proofs of the other lemmas concerning `lookup` involve several intermediate steps the description of which would not explain new features of program verification with Lego. We therefore omit them here. The interested reader is, however, invited to examine the proofs using ftp'ed copies of the source files.

6.4 Implementation of the functors `SeqDup` and `SeqPairs`

In the definition of the signature `SEQPAIRS` the value of `(hd nil)` remained unspecified. This ensured that our implementation did not actually make use of this value. Yet for the implementation of `SeqPairs` we are forced to give a meaning to `(hd nil)`. So what we have to do is to change our original specification by augmenting the arguments to `SeqPairs` by a "proof" that the underlying type of `Ident` is nonempty, which we shall use as default value together with `null_attrib`. Admittedly this problem could be solved more elegantly by introducing exceptions.

```
Goal {Ident:IDENT}{Attrib:ATTRIB} (Ident.1) -> (SEQPAIRS Ident Attrib);
   See the Lego code for this straightforward implementation
Save SeqPairs;
```

Another solution to this problem would be to assume every type to be non-empty, i.e. to contain an error element, but then one would have to show at each stage that one never actually uses these default values because they lack a computational meaning. A similar problem arises with the use of exceptions where one must prove that no exception can arise provided all inputs meet their specification.

The implementation of `SeqDup` poses no further problems.

```
Goal {Ident:IDENT}{Attrib:ATTRIB}{Seq:SEQPAIRS Ident Attrib}
   (SEQDUP Ident Attrib Seq);
   ...
Save SeqDup;
```

6.5 Putting things together

Now all functors are implemented; we may apply them to actual instances of `IDENT` and `ATTRIB`. As an example we choose the natural numbers for both:

```
[Ident_nat = (nat,eq_nat,eqtype_nat):IDENT];
[Attrib_nat = (nat,eq_nat,zero,eqtype_nat):ATTRIB];
```

where `eq_nat` is the equality function on `nat` defined in Section 3 and `eqtype_nat` is a proof of its correctness (which we must, of course, construct). We obtain a symbol table as follows

```
[Symtab = [seqpairs=SeqPairs Ident_nat Attrib_nat zero]
         (ChangeAttrib Ident_nat Attrib_nat
         (Symtab' Ident_nat Attrib_nat seqpairs
         (SeqDup Ident_nat Attrib_nat seqpairs)))];
```

By unpacking the obtained structure

```
[symtab1=Symtab.1]; [empty1=Symtab.2.1];
[add1=Symtab.2.2.1]; [change_attrib1=Symtab.2.2.2.1];
[present1=Symtab.2.2.2.2.1]; [lookup1=Symtab.2.2.2.2.2.2.2.2.1];

[one=succ zero];[two=succ one];[three=succ two];
[four=succ three];[five=succ four];[six=succ five];
[thesymtab=(add1 one three
(add1 one four
(add1 two five
(add1 three two empty1))))];
```

thesymtab $= (\langle \mathrm{ident}, \texttt{attrib}\rangle, \langle 3,2\rangle, \langle 2,5\rangle, \langle 1,3\rangle)$

we can run some "pseudo calculations" as follows:

```
Goal Q three (lookup1 one thesymtab);
Refine Q_refl;
** QED ***
Goal isnotthecase (present1 four thesymtab);
Refine Q_refl;
** QED ***
```

An actual calculation by normalisation as shown in Section 3 would in principle be possible but extremely time and space consuming, since all the proof terms would be normalised, too. Of course in practice one would not run the actual Lego program but an equivalent version implemented in a pure programming language.

7 Conclusions and related work

We have presented a method to develop programs from specifications together with machine checked proofs of their correctness. In the example we have demonstrated the practical use of this method. Compared to the case studies done with "logical framework" systems like HOL [11] ours is rather modest. We see, however, two advantages of the described approach. First, since we derive all proof rules internally, we automatically obtain consistency proofs for them. E.g. in the case of "spantypes" or terminating recursion this is not obvious. Secondly, as we can think of ECC as the internal language of some categorical structure, our syntactic constructions give rise to a categorical denotational semantics of program development.

Different in another sense is the idea of developing programs in type theory using *program extraction* [23, 7]. Instead of — as we have described — constructing both a program and a correctness proof in parallel, here a program is automatically extracted from a constructive proof of its existence. Although appealing from a theoretical point of view, we believe that this approach is less intuitive and also less likely to scale up to larger examples, since it appears much more difficult to prove the existence of a program rather than to prove a given program correct.

Much work remains to be done. Apart from developing larger case studies and supplying a more user–friendly interface the following extensions suggest themselves.

- If one wants to stick to the restriction that all functions have to be total, then better means to incorporate general recursion have to be found. We think of a generalisation of the described recursor to arbitrary well–founded orderings as complexity measures.

- To deal with more complex specifications like, e.g. sets which cannot simply be implemented as a free algebra, we need a notion of *quotient type* which would also treat the problem of observational equivalence [26].
- The extension of our approach by computational monads [22] such as exceptions and side effects should be investigated. Here appropriate proof principles have to be stated and proved.
- Since, as we believe, the main value of machine–supported development and verification lies in the treatment of concurrent systems, an application of type–theoretic methods to this area would be highly desirable. Yet this attempt is at the moment much less concrete than the other extensions.

Acknowledgements

First of all I want to thank my supervisor Don Sannella who taught me to put my clumsy ideas into (hopefully) correct English, suggested many improvements and also patiently read several drafts of this paper. Next I should like to thank Terry Stroup, the supervisor of my *Diplomarbeit*, who has provided a wonderful working environment in Erlangen and helped me to overcome the first difficulties. Quite influential in this work were many fruitful discussions with Thorsten Altenkirch. Thanks also to Zhaohui Luo who helped with proof reading earlier draft. Finally I want to thank the members of the Edinburgh Lego Club who patiently listened to two talks on the subject.

References

[1] Thorsten Altenkirch. Impredicative representations of categorical datatypes, thesis proposal. University of Edinburgh, 10 1990. unpublished.

[2] Corrado Boehm and Alessandro Berarducci. Automatic Synthesis of typed λ-programs on Term Algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[3] R. Burstall and B. Lampson. Pebble, a kernel language for modules and abstract datatypes. In *Symposium Semantics of Data Types*. Springer, LNCS 173, 1984.

[4] Rod Burstall. Computer assisted proof for mathematics: anLEGO proof system. Technical Report ECS-LFCS-91-132, University of Edinburgh, January 1991.

[5] Rod Burstall and Joseph Goguen. The semantics of CLEAR, a specification language. In Dines Björner, editor, *Abstract Software Specifications*, pages 292–332. Springer, 1980. LNCS 86.

[6] Rod Burstall and James McKinna. Deliverables: An approach to program semantics in constructions. Technical report, LFCS, 1990.

[7] Robert Constable and D. J. Howe. Nuprl as a general logic. In *Logic and Computer Science*, pages 123–196. Academic Press, 1990.

[8] Thierry Coquand. Metamathematical investigations of a calculus of constructions. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 91–118. Academic Press Ltd., 1990.

[9] Gilles Dowek et al. The COQ proof assistant user's guide, V5.6. Rapport technique 134, INRIA Rocquencourt, Dec. 1991.

[10] Jordi Farrés-Casals. Proving correctness w.r.t. specifications with hidden parts. In *Algebraic and Logic Programming*, pages 25–39. Springer, 1990. LNCS 463.

[11] M. Gordon. HOL: A proof generating system for Higher Order Logic. In *VLSI Specification, verification, and synthesis*. Kluwer Academic Publishers, 1988.

[12] Robert Harper, David MacQueen, and Robin Milner. *Standard ML*. LFCS Education. University of Edinburgh, 1987. A five day course *Functional Programming in ML*; Part I: The Standard ML Core Language (Milner); Part II: Standard ML Input/Output (Harper);.

[13] Ian Hayes, editor. *Specification Case Studies*. Prentice–Hall, 1987.

[14] Martin Hofmann-Fischer. *Verifikation von ML-Programmen mit dem Beweisprüfer Lego*. Diplomarbeit, Universität Erlangen, 1991.

[15] Gérard Huet. A uniform approach to type theory. Rapports de Recherche 795, INRIA, Fevrier 1988.

[16] Bernd Krieg-Brückner and Donald Sannella. Structuring specifications in-the-large and in-the-small: Higher-order functions, dependent types and inheritance in SPECTRAL. Technical Report ECS-LFCS-91-135, University of Edinburgh, January 1991.

[17] Zhaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, July 1990.

[18] Zhaohui Luo. Program specification and data refinement in type theory. Technical report, LFCS, 1991.

[19] Zhaohui Luo and Randy Pollack. LEGO Proof Development System: User's Manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.

[20] David MacQueen. Using dependent types to express modular structure. In *13th ACM Symposium on Principles of Programming Languages*, pages 277–286. ACM, 1986.

[21] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis·Napoli, 1984.

[22] Eugenio Moggi. Computational lambda calculus and monads. Technical report, University of Edinburgh, November 1988. preprint.

[23] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löfs Type Theory, An Introduction.* Clarendon Press, Oxford, 1990.

[24] Donald Sannella. Formal program development in Extended ML for the working programmer. Report ECS-LFCS-89-102, University of Edinburgh, December 1989.

[25] Donald Sannella and Andrzej Tarlecki. Extended ML: An institution-independent framework for formal program development. In *Category Theory and Computer Programming*, pages 364–389. Springer, 1986. LNCS 240.

[26] Donald Sannella and Andrzej Tarlecki. On Observational Equivalence and Algebraic Specification. *Journal of Computer and System Sciences*, 34:150–178, 1987. extended abstract in LNCS 185, 308–322, 1985.

[27] Donald Sannella and Andrzej Tarlecki. Extended ML: Past, present and future. Technical Report ECS-LFCS-91-138, University of Edinburgh, February 1991.

[28] Paul Taylor. Playing with LEGO: Some Examples of Developing Mathematics in the Calculus of Constructions. Technical Report ECS-LFCS-89-89, University of Edinburgh, August 1989.

8 Appendix

The Lego files can be obtained on e-mail request (mxh@dcs.ed.ac.uk) or by anonymous internet ftp from host `ftp.dcs.ed.ac.uk` (address 129.215.160.5), directory `pub/dts`, file `mxh-lego.tar.Z` (this is a compressed tar file which contains a directory including a file `README` for further information). Here is a sample dialog:

```
ftp ftp.dcs.ed.ac.uk
Name:  anonymous
Password:  your email-address
ftp> binary
ftp> cd pub/dts
ftp> get mxh-lego.tar.Z
ftp> bye
```

After the file is transferred it should be uncompressed and then extracted using `tar`. For example: `zcat mxh-lego.tar.Z | tar fox -`.