Implementing a Category of Sets in ALF

Peter Dybjer*and Verónica Gaspes†
Chalmers University of Technology and University of Gothenburg

September 16, 1994

1 Introduction

Peter Aczel [1] and Gerard Huet [8] have implemented the category of sets in LEGO and Coq respectively. Here we show an implementation of the category of sets in *ALF* [2], a proof assistant based on *Martin-Löf's logical framework* (or *theory of logical types*) [10].

We used WINDOW ALF. This system allows one to manipulate the proof term in order to refine it until it is complete. Some facilities are provided which show the term in a readable way (special symbols for constants, infix use of symbols, hiding of arguments, etc). What is presented below is, unfortunately, not what is shown on the screen, but the source code for the type checker and the window interface. Thus, for instance, no arguments are hidden and `lambda` is used instead of λ. We refer to the introduction to the ALF chapter of the library for further information.

We have essentially followed Peter Aczel's development. But we have also introduced some modifications which seemed natural with respect to ALF's formalism. The most significant differences concern the question of level (or size) which is discussed in the last section.

Like Aczel's, our notion of *category* has an explicit equivalence relation on each hom-set as part of the structure, and also an explicit proof that composition preserves this equivalence relation. Similarly, our notion of *setoid* (small setoids will be the objects of our category) carries an explicit equivalence relation with it, and our notion of *map* (arrow of the category of small setoids) carries a proof that it preserves this equivalence relation. The term *setoid*[1] is used to distinguish this notion of set from the basic notion of *set* in Martin-Löf's type theory (and reserved word in ALF!). So a setoid is a set with an equivalence relation on it.

Aczel uses LEGO's universe hierarchy $Type_i$ and universe polymorphism (but not the impredicative type of propositions). His setoids (which he calls sets) are triples $\langle T, R, pr\rangle$ where T is an object in $Type$, R is a relation and pr is a proof that R is an equivalence relation. He then defines the setoid of maps between two setoids. Moreover, his notion of a category has a type of objects and for each pair of objects a hom-setoid. The level of a category depends on the level of the type of objects and the level of the hom-setoid.

*peterd@cs.chalmers.se
†vero@cs.chalmers.se
[1]We found this term in a paper by Martin Hofmann [7], who in turn credits it to Rod Burstall.

In contrast, ALF has no preimplemented universe hierarchy. There is the notion of type and the basic type of sets available. In addition it provides a facility for defining *contexts*, and also for defining *substitutions*.

A context Γ is a list

$$[x_1 : \alpha_1; \ldots; x_n : \alpha_n]$$

of pairs of variables and types.

A substitution γ is a list of assignments

$$\{x_1 := a_1; \ldots; x_n := a_n\}$$

of objects to the variables. We write

$$\gamma : \Gamma$$

to express that the substitution γ fits the context Γ, that is, that that the variables coincide and the assigned objects have the appropriate types: $a_1 : \alpha_1, \ldots, a_n : \alpha_n$

Contexts are suitable for representing algebraic notions [3, 5, 11, 6]. For instance, the notion of a group can be represented by a context specifying the types of its components, and then a particular group can be represented by a substitution fitting this context.

Contexts can implement arbitrary first order theories, generalized algebraic theories [4], etc.

2 Implementation of the notion of a category

We define the notion of a category as a context CATEGORY:

```
CATEGORY is [Ob:Set;
             Hom:(Ob)(Ob)Set;
             comp:(A,B,C:Ob)(g:Hom(B,C))(f:Hom(A,B))Hom(A,C);
             id:(A:Ob)Hom(A,A);

             EqHom:(A,B:Ob)(Hom(A,B))(Hom(A,B))Set;
             eq_hom_is_equiv:(A,B:Ob)Equivalence(Hom(A,B),EqHom(A,B));
             eq_hom_is_congr:(A,B,C:Ob)Congruence(Hom(B,C),Hom(A,B),Hom(A,C),
                                                 EqHom(B,C),EqHom(A,B),EqHom(A,C),
                                                 comp(A,B,C));

             assoc:(A,B,C,D:Ob)(h:Hom(C,D))(g:Hom(B,C))(f:Hom(A,B))
                   EqHom(A,D,
                         comp(A,C,D,h,comp(A,B,C,g,f)),
                         comp(A,B,D,comp(B,C,D,h,g),f));
             left_id_law:(A,B:Ob)(f:Hom(A,B))
                         EqHom(A,B,comp(A,B,B,id(B),f),f);
             right_id_law:(A,B:Ob)(f:Hom(A,B))
                          EqHom(A,B,comp(A,A,B,f,id(A)),f)]
```

We have separated the context into three parts. The first part declares the signature. Both objects and hom-sets are required to form sets in the sense of Martin-Löf's type theory. The second part declares an explicit relation on the hom-sets which is required to be an equivalence preserved by composition. The third part declares the category axioms.

We have used two auxiliary definitions. Firstly

```
Equivalence   : (A:Set;R:(A;A)Set)Set     []    C
  equivalence : (A:Set;R:(A;A)Set;
                 refl:(x:A)R(x,x);
                 symm:(x:A;y:A;R(x,y))R(y,x);
                 trans:(x:A;y:A;z:A;R(x,y);R(y,z))R(x,z))
                Equivalence(A,R)     []    C
```

This is an inductive definition: `equivalence` is a constructor accepting three arguments in addition to the *parameters* `A` and `R` (alternatively we could have used an explicit definition refering to the predefined set of tuples). This defines the set of proofs that `R` is an equivalence relation. Similarly

```
Congruence    : (A,B,C:Set)
                (R1:(A)(A)Set)
                (R2:(B)(B)Set)
                (R3:(C)(C)Set)
                (op:(A)(B)C)
                Set [] C
  congruence : (A,B,C:Set)
                (R1:(A)(A)Set)
                (R2:(B)(B)Set)
                (R3:(C)(C)Set)
                (op:(A)(B)C)
                (cong:(a,a':A)(b,b':B)(R1(a,a'))(R2(b,b'))
                      R3(op(a,b),op(a',b')))
                Congruence(A,B,C,R1,R2,R3,op)  [] C
```

3 The category of natural numbers

As a simple example we implement the category which has the natural numbers as objects and where there is a single arrow between m and n provided $m \leq n$. In type theory, we express this by taking the arrows between m and n to be the proofs of $m \leq n$, and stating that all proofs of $m \leq n$ are equal.

```
NATURALNUMBERS
          is {Ob := N;
              Hom := Leq;
              comp := Leq_trans;
              id := Leq_ref;

              EqHom := SingleArrows;
              eq_hom_is_equiv := sa_is_equiv;
              eq_hom_is_congr := preservation;
              assoc := trans_is_assoc;
              left_id_law := left_law_leq_ref;
              right_id_law := right_law_leq_ref}

          : CATEGORY []
```

This substitution was built from proofs in the library of ALF about the natural numbers and the ordering relation. The equivalence relation `SingleArrows` equates all proofs of inequality between two numbers:

```
SingleArrows = [n,m,f1,f2]N1 : (n:N;m:N;f1:Leq(n,m);f2:Leq(n,m))Set  []
```

4 The category of setoids

We now give the substitution that implements the category of small setoids. When defining the different components we use the following constants from the ALF library: `U`, the universe whose elements are codes for small sets; `T`, the function decoding elements of `U` into sets; `Function`, the set of functions between two sets and its dependent version `Pi`; `lambda` and `Lambda`, ordinary and dependent function abstraction; `apply` and `Apply`, ordinary and dependent function application.

```
        SETOID
             is {Ob := Setoid;
                 Hom := Map;
                 comp := o;
                 id := id_map;

                 EqHom := Ext;
                 eq_hom_is_equiv := ext_is_equiv;
                 eq_hom_is_congr := ext_is_congr;

                 assoc := o_is_assoc;
                 left_id_law := id_map_is_left_id;
                 right_id_law := id_map_is_right_id}

             : CATEGORY []
```

The set of objects is that of small setoids. Its elements are triples: a small set, a small binary relation and a proof that this relation is an equivalence.

```
Setoid  : Set     []    C
 setoid : (el:U;
          eq:(T(el);T(el))U;
          pr:Small_Equivalence(el,eq))
         Setoid     []    C
```

It has projections:

```
el : (Setoid)U     []    I
el(setoid(el1,eq,pr1)) = el1

eq : (A:Setoid;T(el(A));T(el(A)))U     []    I
eq(setoid(el1,eq1,pr1),h,h1) = eq1(h,h1)

prf : (A:Setoid)Small_Equivalence(el(A),eq(A))     []    I
prf(setoid(el1,eq1,pr1)) = pr1
```

Note that `Small_Equivalence` only applies to small relations and can be defined using `Equivalence` above.

Arrows are functions on the underlying sets of setoids which preserve the equivalence relation on these setoids:

```
Map  : (A:Setoid;B:Setoid)Set     []    C
 map : (A:Setoid;B:Setoid;
        f:Function(T(el(A)),T(el(B)));
        pres:(x:T(el(A));
              y:T(el(A));
              T(eq(A,x,y)))
             T(eq(B,
                  apply(T(el(A)),T(el(B)),f,x),
                  apply(T(el(A)),T(el(B)),f,y))))
       Map(A,B)     []    C
```

We have the projections:

```
fun : (A:Setoid;B:Setoid;
       Map(A,B))
      Function(T(el(A)),T(el(B)))     []    I
   fun(A,B,map(_,_,f,pres)) = f

pres : (A:Setoid;B:Setoid;
        f:Map(A,B);
        x:T(el(A));
        y:T(el(A));
        T(eq(A,x,y)))
       T(eq(B,
            apply(T(el(A)),T(el(B)),fun(A,B,f),x),
            apply(T(el(A)),T(el(B)),fun(A,B,f),y)))     []    I
   pres(A,B,map(_,_,f1,pres1),x,y,h) = pres1(x,y,h)
```

Composition of maps is composition of the underlying functions:

```
o =
  [A,B,C,g,f]map(A,C,
                 lambda(T(el(A)),T(el(C)),
                        [h]apply(T(el(B)),T(el(C)),
                                 fun(B,C,g),
                              apply(T(el(A)),T(el(B)),
                                    fun(A,B,f),
                                    h))),
                [x,y,h]pres(B,C,
                           g,
                           apply(T(el(A)),T(el(B)),fun(A,B,f),x),
                           apply(T(el(A)),T(el(B)),fun(A,B,f),y),
                           pres(A,B,f,x,y,h)))
  :
  (A:Setoid;B:Setoid;C:Setoid;g:Map(B,C);f:Map(A,B))Map(A,C)     []
```

The identity map:

```
id_map =
   [A]map(A,A,lambda(T(el(A)),T(el(A)),[h]h),[x,y,h]h)
   :
   (A:Setoid)Map(A,A)     []
```

Equality of arrows is extensional equality of maps:

```
Ext =
    [A,B,f,g]Pi(T(el(A)),
               [h]T(eq(B,
                       apply(T(el(A)),T(el(B)),fun(A,B,f),h),
                       apply(T(el(A)),T(el(B)),fun(A,B,g),h))))
    :
    (A:Setoid;B:Setoid;f:Map(A,B);g:Map(A,B))Set     []
```

The proof that this is an equivalence relation:

```
ext_is_equiv =
  [A,B]equivalence(Map(A,B),
                   Ext(A,B),
                   [x]Lambda(T(el(A)),
                             [h]T(eq(B,
                                     apply(T(el(A)),T(el(B)),fun(A,B,x),h),
                                     apply(T(el(A)),T(el(B)),fun(A,B,x),h))),
                             [x']pr_refl(el(B),
                                         eq(B),
                                         prf(B),
                                         apply(T(el(A)),T(el(B)),fun(A,B,x),x'))),
                   [x,y,h]Lambda(T(el(A)),
                                 [h']T(eq(B,
                                          apply(T(el(A)),T(el(B)),fun(A,B,y),h'),
                                          apply(T(el(A)),T(el(B)),fun(A,B,x),h'))),
                                 [x']pr_symm(el(B),
                                             eq(B),
                                             prf(B),
                                             apply(T(el(A)),T(el(B)),fun(A,B,x),x'),
                                             apply(T(el(A)),T(el(B)),fun(A,B,y),x'),
                                             Apply(T(el(A)),
                                                   [h']T(eq(B,
                                                            apply(T(el(A)),T(el(B)),
                                                                  fun(A,B,x),h'),
                                                            apply(T(el(A)),T(el(B)),
                                                                  fun(A,B,y),h'))),
                                                   h,
                                                   x'))),
                   [x,y,z,h,h1]Lambda(T(el(A)),
                                      [h']T(eq(B,
                                               apply(T(el(A)),T(el(B)),
                                                     fun(A,B,x),h'),
                                               apply(T(el(A)),T(el(B)),
                                                     fun(A,B,z),h'))),
                                      [x']pr_trans(el(B),
                                                   eq(B),
                                                   prf(B),
                                                   apply(T(el(A)),T(el(B)),
                                                         fun(A,B,x),x'),
                                                   apply(T(el(A)),T(el(B)),
                                                         fun(A,B,y),x'),
                                                   apply(T(el(A)),T(el(B)),
                                                         fun(A,B,z),x'),
                                                   Apply(T(el(A)),
                                                         [h']T(eq(B,
                                                     apply(T(el(A)),
                                                           T(el(B)),
                                                           fun(A,B,x),
                                                           h'),
                                                     apply(T(el(A)),
                                                           T(el(B)),
                                                           fun(A,B,y),
                                                           h'))),
                                            h,
                                            x'),
                                      Apply(T(el(A)),
                                            [h']T(eq(B,
                                                     apply(T(el(A)),
                                                           T(el(B)),
                                                           fun(A,B,y),
                                                           h'),
                                                     apply(T(el(A)),
                                                           T(el(B)),
                                                           fun(A,B,z),
                                                           h'))),
                                            h1,
                                            x'))))
:
(A:Setoid;B:Setoid)Equivalence(Map(A,B),Ext(A,B))     []
```

The proof that composition preserves extensional equality:

```
ext_is_congr =
 [A,B,C]congruence(Map(B,C),
                  Map(A,B),
                  Map(A,C),
                  Ext(B,C),
                  Ext(A,B),
                  Ext(A,C),
                  o(A,B,C),
                  [g,g',f,f',geqg',feqf']
                     Lambda(T(el(A)),
                            [h]T(eq(C,
                                    apply(T(el(A)),T(el(C)),
                                          fun(A,C,o(A,B,C,g,f)),h),
                                    apply(T(el(A)),T(el(C)),
                                          fun(A,C,o(A,B,C,g',f')),h))),
                            [x]pr_trans(el(C),
                                        eq(C),
                                        prf(C),
                                        apply(T(el(A)),T(el(C)),
                                              fun(A,C,o(A,B,C,g,f)),x),
                                        apply(T(el(B)),T(el(C)),
                                              fun(B,C,g),
                                              apply(T(el(A)),T(el(B)),
                                                    fun(A,B,f'),x)),
                                        apply(T(el(A)),T(el(C)),
                                              fun(A,C,o(A,B,C,g',f')),x),
                                        pres(B,C,
                                             g,
                                             apply(T(el(A)),T(el(B)),
                                                   fun(A,B,f),x),
                                             apply(T(el(A)),T(el(B)),
                                                   fun(A,B,f'),x),
                                             Apply(T(el(A)),
                                                   [h]T(eq(B,
                                                        apply(T(el(A)),T(el(B)),
                                                              fun(A,B,f),h),
                                                        apply(T(el(A)),T(el(B)),
                                                              fun(A,B,f'),h))),
                                                   feqf',
                                                   x)),
                                         Apply(T(el(B)),
                                               [h]T(eq(C,
                                                       apply(T(el(B)),T(el(C)),
                                                             fun(B,C,g),h),
                                                       apply(T(el(B)),T(el(C)),
                                                              fun(B,C,g'),h))),
                                                 geqg',
                                                 apply(T(el(A)),T(el(B)),
                                                       fun(A,B,f'),x)))))
          :
          (A,B,C:Setoid)Congruence(Map(B,C),
                                   Map(A,B),
                                   Map(A,C),
                                   Ext(B,C),
                                   Ext(A,B),
                                   Ext(A,C),
                                   o(A,B,C)) []
```

The proof that composition is associative:

```
o_is_assoc =
        [A,B,C,D,h,g,f]Lambda(T(el(A)),
                              [h']T(eq(D,
                                      apply(T(el(A)),T(el(D)),
                                            fun(A,D,o(A,C,D,h,o(A,B,C,g,f))),
                                            h'),
                                      apply(T(el(A)),T(el(D)),
                                            fun(A,D,o(A,B,D,o(B,C,D,h,g),f)),
                                            h'))),
                              [x]pr_refl(el(D),
                                         eq(D),
                                         prf(D),
                                         apply(T(el(A)),T(el(D)),
                                         fun(A,D,o(A,C,D,h,o(A,B,C,g,f))),
                                         x)))
        :
        (A:Setoid;B:Setoid;C:Setoid;D:Setoid;
         h:Map(C,D);g:Map(B,C);f:Map(A,B))
        Ext(A,D,o(A,C,D,h,o(A,B,C,g,f)),o(A,B,D,o(B,C,D,h,g),f))     []
```

The proofs that the identity map satisfies the identity laws with respect to composition:

```
id_map_is_left_id  =
            [A,B,f]Lambda(T(el(A)),
                          [h]T(eq(B,
                                  apply(T(el(A)),T(el(B)),
                                        fun(A,B,o(A,B,B,id_map(B),f)),
                                        h),
                                  apply(T(el(A)),T(el(B)),fun(A,B,f),h))),
                          [x]pr_refl(el(B),
                                     eq(B),
                                     prf(B),
                                     apply(T(el(A)),T(el(B)),
                                           fun(A,B,o(A,B,B,id_map(B),f)),
                                           x)))
            :
            (A:Setoid;B:Setoid;f:Map(A,B))Ext(A,B,o(A,B,B,id_map(B),f),f)      []

id_map_is_right_id =
             [A,B,f]Lambda(T(el(A)),
                           [h]T(eq(B,
                                   apply(T(el(A)),T(el(B)),
                                         fun(A,B,o(A,A,B,f,id_map(A))),
                                         h),
                                   apply(T(el(A)),T(el(B)),fun(A,B,f),h))),
                           [x]pr_refl(el(B),
                                      eq(B),
                                      prf(B),
                                      apply(T(el(A)),T(el(B)),
                                            fun(A,B,o(A,A,B,f,id_map(A))),
                                            x)))
             :
             (A:Setoid;B:Setoid;f:Map(A,B))Ext(A,B,o(A,A,B,f,id_map(A)),f)      []
```

5 Discussion

Our notion of category is the “largest” we can define as a context, with objects and hom-sets forming sets. This is analogous to MacLane [9], who requires these components to be sets in the classical sense. If we call this notion *large* category, we can also define a notion of *small* category with a small set of objects and small hom-sets:

```
SMALL_CATEGORY is
            [Ob:U;
             Hom:(T(Ob))(T(Ob))U;
             ... ]
```

To form the category of small categories we have to define the set of small categories as a set of tuples:

```
Small_category : Set

small_category : (Ob:U;
                  Hom:(T(Ob))(T(Ob))U;
                  ...)
                 Setoid
```

Then we can define the set of functors between small categories, and the other components needed to form a category.

Alternatively, we could also define a notion of category which has a *type* of objects (but hom-*sets*). In this way we could speak of the category of *all* (basic) sets or *all* setoids, not just the small ones. But in the present ALF-formalism it is not possible to express this notion as a context. (What we can do however is to prove a sequence of propositions corresponding to constructing the category of all set(oid)s.)

This can be compared to the classical definition of a category as having *classes* of objects and arrows, which is used in some texts as an alternative to MacLane's definition.

References

[1] Peter Aczel. Galois: A theory development project. A report on work in progress, for the Turin meeting on the Representation of Mathematics in Logical frameworks, 1993.

[2] L. Augustsson, T. Coquand, and B. Nordström. A short description of another logical framework. In *Proceedings of the First Workshop on Logical Framewrks, Antibes*, pages 39–42, 1990.

[3] Gustavo Betarte. Licentiate thesis. draft, 1993.

[4] John Cartmell. Generalized algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, 32:209–243, 1986.

[5] Catarina Coquand. Analysis of simply typed lambda-calculus in ALF. In *Draft Proceedings of the Winter Meeting, Tanum Strand*, 1993.

[6] Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. Draft paper, 1993.

[7] Martin Hofmann. Elimination of extensionality and quotient types in Martin-Löf's type theory. Draft paper, May 1993.

[8] Gerard Huet. 1993.

[9] Saunders Mac Lane. *Categories for the Working Mathematician.* Springer-Verlag, 1971.

[10] Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's Type Theory: an Introduction.* Oxford University Press, 1990.

[11] Alvaro Tasistro. Extension of martin-löf's theory of types with labelled record types and subtyping. Talk presented at the workshop Types for Proofs and Programs, Nijmegen, May 1993.