CATEGORY THEORY
as an extension of
Martin-Löf Type Theory.

Roy Dyckhoff
Department of Computational Science,
University of St Andrews.

0. Abstract

Category theory has long been widely recognised as being conveniently formalisable in constructive mathematics. We describe a computer implementation of its basic concepts, as an extension of the Göteborg implementation of Martin-Löf's theory of types; we discuss some alternative formulations of the theory (and our reasons for rejecting them) and briefly comment on the problems of automating the theorem-proving facilities of such an implementation.

1. Introduction

Our purposes in implementing such a system are various. First, the ability to prove theorems of category theory would be useful, in connection with the current [33] growth of interest in category theory for formalising notions from computer science, such as implementation of languages, data abstraction, and algorithm development. Second, it is one of the more interesting theories in mathematics, with many different levels of meaning and application, allowing a wide variety of different ways of looking at a single problem, and with important abstract concepts like adjointness. Third, there are links [07], [41] between category theory and strong typing systems (such as that of ML, or of Martin-Löf), which the implementation of categorical concepts should help us to explore. Fourth, Martin-Löf type theory is now being investigated by many as an abstract functional programming language, and the implementation of category theory therein presented an opportunity further to develop the type theory system itself.

Rydeheard and Burstall [06] have implemented a more substantial part of category theory. Our approach differs from theirs by also covering equations. Put otherwise, 'category' in our system is a much stronger notion than theirs, just as there is a difference between types of Martin-Löf's system and those of ML.

2. Background on Martin-Löf type theory

Martin-Löf type theory [25],[26],[27] is a particularly suitable foundation for category theory, for a number of reasons (apart from the constructive nature of category theory.) We refer the reader to [27], or to Beeson [02], for a full presentation of the theory, roughly as we require it. Quite a lot of what we do should make sense to any reader acquainted with any natural deduction system. For such readers, it should only be necessary to add that to declare a variable of some type A, it is necessary to have first proved, or to have assumed, that the type expression denoting A is well-formed. We mention now some of our reasons for choosing Martin-Löf's theory as a foundation.

First, the identification of propositions with types allows, here as elsewhere, a smooth and unified treatment of two common parts of an argument: viz, the declaration of a typed variable (e.g. "let **C** be a category") and the assumption of an equation (e.g. "suppose $f \cdot h = g \cdot h$", where the infix operator symbol $\cdot$ denotes composition.)

Second, one of its underlying principles, that every term of the theory has a type, (and thus that quantification is only permissible over a type) fits conveniently with, for example, the common supposition in category theory

"let $f : X \to Y$ be a morphism of the category **C**",

which translates into the last item of the assumption list

"[C:CAT ; X:Ob(C) ; Y:Ob(C) ; f:Mor(C,X,Y)]".

Third, as the above example also shows, it is common for the types in assumptions in category theory to depend on variables introduced in earlier assumptions; assumption lists are therefore lists (or at least, partially ordered sets) rather than, as in ordinary predicate calculus, merely sets. This however is an important feature of Martin-Löf's theory, and comes with obvious restrictions on the discharge of assumptions without discharging other assumptions dependent thereon.

Fourth, the partiality of the composition operator can conveniently be managed by considering it as having the dependent function type

$$\Pi(A,B,D,E : Ob(C),\ [Mor(C,A,B)\ \#\ Mor(C,B,D) \rightarrow Mor\ (C,A,D)]\ )$$

(in a modification of the Göteborg notation for Martin-Löf type theory.) In fact, we adopt a slightly different approach, allowing a more convenient notation. Similar observations apply of course to many of the other operators, such as the construction of a pullback square from a pair of morphisms with common codomain.

Fifth, its hierarchy of universes U0,U1,U2,... is convenient for handling the problems of constructing "the category of (small) categories", "the 'category' of categories", ... Type theory has, unless carefully formulated, similar problems, avoided by means of this hierarchy. The rules for (small) category theory outlined below can easily be adapted for larger categories.

The Göteborg implementation (which we call the *GTTS*, for the Göteborg Type Theory System) of this theory, designed and programmed by Petersson [31],[32], is written in Franz-Lisp and Edinburgh ML [17], and runs (in our case, on a VAX 11/750) under the Unix operating system. ML allows the definition of an abstract data type 'thm' whose elements denote judgments, alias theorems, of the theory, and whose operations represent the inference rules of the theory. The abstract nature of this data type officially prohibits invalid constructions of theorems; as currently implemented, however, the system permits the addition of new type constructors and new inference rules, without redefinition of the type 'thm', and this approach has been followed in the present case.

The NuPRL system [09] at Cornell implements a type theory essentially the same as Martin-Löf's. Our implementation could instead have been done using NuPRL: the GTTS was however more convenient, in allowing the programming to be done in ML rather than in LISP.

In our extension of the type theory system, we follow the traditional pattern by giving, for each new type constant or type constructor, a group of rules, divided into the formation rules, the introduction rules, the elimination rules, and the computation rules. Most of these come in two forms, a simple rule and a corresponding 'extensionality' rule: to save space, we have omitted to detail the extensional versions of the rules, but the reader familiar with such rules can easily work them out. We have been careful to avoid rules requiring superfluous premisses: official presentations ([27], [02]) of the theory generally omit to mention many premisses explicitly except that the reader is invited to guess where and what they should be. In practice, many of these premisses are unnecessary, and such unnecessary premisses are not required in the GTTS. (In one or two cases, e.g. with the rule SIGMAintr, necessary premisses too are discarded!) We follow this simplification of the theory: a fuller discussion appears in [14].

3. Relation to other work

Burstall and Rydeheard [06],[37],[38] have implemented a substantial number of concepts and constructions of category theory in standard ML (originally, in HOPE). Their purpose is to have a framework in which categorical constructions can be done (with a view to applications to program development, and algorithm specification); but it does not include the equations satisfied by category theory, nor, *a fortiori*, admit of formal verifications that the constructions (e.g. of

finite products of objects) have the desired properties. Moreover, their representation cannot fully express the partiality of composition. Provided the structured collections of objects and morphisms manipulated really are categories, no harm appears to be done. In the long run, however, the need for such verifications will arise: for example, at the interface with formal methods in other theories, such as universal algebra.

In the work described in this report, equations and verifications (that certain structures are categories, functors, ...) are fundamental. This distinction is reflected in the following: 'category' is defined as a datatype of the meta-language (HOPE, or Standard ML) by Burstall and Rydeheard, but, in our case, as a type of the object language (type theory, which happens to be manipulated by the meta-language ML). Martin-Löf type theory has a much richer type structure than that of ML, hence we are able to manipulate, for example, assumptions of equations by considering the equations as (logical) formulae and thus as types.

Many others, (notably Barr & Wells [01], Benecke & Reichel[03], Burmeister [04], Cartmell [07], [08], Coquand & Huet [10], [11], Curien [12], Freyd [16], Hamza [20], Huet [22], Lambek & Scott [23], Poigné [35], [36], Scott [40], Szabo [42], and Watjen & Struckmann [43]) have represented notions from category theory in various formal languages (although, with the exception of those of Curien, Coquand & Huet, Watjen & Struckmann, and Hamza's implementation of Szabo [42], none of these representations appear to have been implemented.) Some of these representations have been in a form designed only for equational reasoning, and although it is arguable that all categorical reasoning is ultimately equational, or at least that much of it can be so expressed, the equational format is often fairly unnatural. For example, existence of pullbacks cannot be expressed (in an equational framework) without designing the notation so that all pairs of morphisms with a common codomain have a pullback, which is canonical: a rather severe restriction.

The closest approach to ours appears to be that of Coquand & Huet [11], who axiomatise the notion of category in the language of constructions, (a blend of Martin-Löf's non-extensional type theory [25] and the second-order theory of Girard); their definition of category is an axiomatic one, similar to one of those in our chapter 6, and presumably suffering from the same problems.

4. Remarks on notation

In presenting the rules for the type constant CAT and the type operators FUNC, and NAT, we adopt the terminology of the Göteborg system. Expressions of the object language, i.e. type theory, usually appear in double quotes. Judgments consist of one of the four sorts of formula

"A type", "A = B", "a : A", "a = b : A"

followed, in square brackets, by an assumption list; these mean, respectively, that "A" denotes a type, that "A" and "B" denote equal types, that "a" denotes an element of type denoted by "A", and that "a" and "b" denote equal elements of the type denoted by "A". Formulae in an assumption list are all of the form "x : A", where "x" is a variable. We use the two levels U (= U0) and U1 of the hierarchy of universes: U is the type of small types, and U1 the next level up this hierarchy. Thus "A : U" formalises the judgment that "A is a small type".

We also use Martin-Löf's notation (introduced in his Munich lectures in 1980) for abstract expressions: a useful explanation is to be found in the book [02] by Beeson. This neatly avoids the use of notation such as B[a/x] to denote "B with a substituted for x", and allows, for example, us to write "$\prod$(A,B)" instead of "$\prod$(x:A, B(x))" or "$\prod$(A, (x)B(x))". The rules for introduction (of categories, for example) become much clearer when this notation is used.

Note also that the Göteborg system uses "#" as the constructor for the product of two types (i.e. conjunction of two propositions).

5. Rules added to the Göteborg type theory system

The basic rules for categories, functors, and natural transformations have been implemented, and are described below in sections 5.1 - 5.3. In fact, the extensional versions of these rules have also been implemented, as noted above. In section 5.0, we illustrate some of the rules for CAT, by giving the 'tree' form of a derivation of the judgment that the relationship of being isomorphic is a reflexive relationship, then an ML program formalising this, and finally a script recording a session with the modified type theory system. In section 5.4, we give an example showing how to define the property of being monomorphic in the language of type theory, and exhibit a proof object for a formula expressing that monomorphisms are closed under composition. In section 5.5, we show how some of the operations on functors and natural transformations, as required in the category of categories, can be defined.

5.0 An example : reflexivity of "Iso"

We use the definition facility of GTTS to abbreviate, by *Iso(C,A,B)*, the propositional formula

```
Sigma(Mor(C,A,B), (x)
    Sigma(Mor(C,B,A), (y)
        #(Eq(Mor(C,A,A), comp(x,y), id),
          Eq(Mor(C,B,B), comp(y,x), id))
        )
     )
```

which asserts the existence of an $x : A \to B$ in C and of a $y : B \to A$ in C such that $x \cdot y$ and $y \cdot x$ equal the identities on A, B respectively.

Here is the derivation in tree form, with the names of the inference rules used on the right. Note that, strictly speaking, PAIRintr needs two premisses, but as they are here the same we save space by omitting the repetition. It is convenient to omit repeated mention of the hypotheses on which each judgment depends: the final judgment clearly depends on the two undischarged assumptions C:CAT, A:Ob(C).

```
                                  ---------- CATform
                                   CAT : U1
                                   ----------  Uelim
                                   CAT type
                                  ---------- VARintr
                                    C : CAT
                                  ---------- CATelim_Ob
                                  Ob(C) : U
                                  ---------- Uelim
                                  Ob(C) type
                                  ---------- VARintr
                                   A : Ob(C)
                                 ---------- CATelim_id
                               id : Mor(C,A,A)
                            ---------------------------- CATelim_IdL
                          comp(id,id) = id : Mor(C,A,A)
                            ------------------------------- EQintr
                        e : Eq(Mor(C,A,A),comp(id,id),id)
                          -------------------------------- PAIRintr
                                pair(e,e) :
                          #(Eq(Mor(C,A,A), comp(id,id),id),
                            Eq(Mor(C,A,A), comp(id,id),id))
                           ---------------------------------- SIGMAintr
                                pair(id,pair(e,e)) :
                                Sigma(Mor(C,A,A),
                          (y)#(Eq(Mor(C,A,A),comp(id,y), id),
                            Eq(Mor(C,A,A),comp(y,id), id)))
                        ------------------------------------------- SIGMAintr
                        pair(id,pair(id,pair(e,e))) : Iso(C,A,A)
```

Thus, we have a proof/element of the proposition/type Iso(C,A,A), constructively showing the truth of this proposition. We give now an ML program representing the above:

```
let CAT = CATform ;;
let CATtype = Uelim CAT ;;
let C = VARintr CATtype "C" ;;
let OC = CATelim_Ob C ;;
let OCtype = Uelim OC ;;
let A = VARintr OCtype "A" ;;
let iA = CATelim_id A ;;
let cii = CATelim_IdL iA ;;
let eE = EQintr cii ;;
let pee = PAIRintr eE eE ;;
let p9 = SIGMAintr
      "(y)#(Eq(Mor(C,A,A),comp(id,y),   id),
            Eq(Mor(C,A,A),comp(y,id),   id))"
       iA pee ;;
let p10 = SIGMAintr
      "(x)Sigma(Mor(C,A,A),(y)
            #(Eq(Mor(C,A,A),comp(x,y),id),
                  Eq(Mor(C,A,A),comp(y,x),id)))"
      iA p9 ;;
```

Here now is a script of the interaction between the GTTS and this ML program; [it has been edited slightly to cope with a bug in the definition facility] :

```
let CAT = CATform ;;
CAT = "CAT : U1" : thm

let CATtype = Uelim CAT ;;
CATtype = "CAT type" : thm

let C = VARintr CATtype "C" ;;
C = "C : CAT [C : CAT]" : thm

let OC = CATelim_Ob C ;;
OC = "Ob(C) : U [C : CAT]" : thm

let OCtype = Uelim OC ;;
OCtype = "Ob(C) type [C : CAT]" : thm

let A = VARintr OCtype "A" ;;
A = "A : Ob(C) [C : CAT; A : Ob(C)]" : thm

let iA = CATelim_id A ;;
iA = "id : Mor(C,A,A) [C : CAT; A : Ob(C)]"
: thm

let cii = CATelim_IdL iA ;;
 cii = "comp(id,id) = id : Mor(C,A,A)
[C : CAT; A : Ob(C)]"
: thm

let eE = EQintr cii ;;
 eE = "e : Eq(Mor(C,A,A),comp(id,id),id)
[C : CAT; A : Ob(C)]"
```

```
: thm

let pee = PAIRintr eE eE ;;
 pee = "pair(e,e)
   : #(Eq(Mor(C,A,A),comp(id,id),id),
   Eq(Mor(C,A,A),comp(id,id),id))
[C : CAT; A : Ob(C)]"
: thm

let p9 = SIGMAintr  "(y)#(Eq(Mor(C,A,A),comp(id,y),   id),
                Eq(Mor(C,A,A),comp(y,id),   id))"
        iA pee ;;
  p9  =  "pair(id,pair(e,e))
 : Sigma(Mor(C,A,A), (y)
       [#(Eq(Mor(C,A,A),comp(id,y),id),
             Eq(Mor(C,A,A),comp(y,id),id))])
[C : CAT; A : Ob(C)]"
: thm

let  p10  =  SIGMAintr "(x)Sigma(Mor(C,A,A),(y)
       #(Eq(Mor(C,A,A),comp(x,y),id),
            Eq(Mor(C,A,A),comp(y,x),id)))"
       iA  p9 ;;
   p10  =  "pair(id,pair(id,pair(e,e)))
: Iso(C,A,A) [C : CAT; A : Ob(C)]"  : thm

Goodbye - 17 seconds of CPU time used
```

5.1 Rules for categories :

The following constants are added to the language:

"CAT", "Cat", "Ob", "Mor", "id", "comp".

They are used as follows : "CAT" denotes the type of small categories, having canonical elements of the form "Cat(O,M,i,c)", where

i) O is a small type (consisting of 'objects'), and

ii) (given objects *X,Y*), *M(X,Y)* is the (small) type of 'morphisms' from *X* to *Y*, and

iii) (given an object X), *i* is the 'identity morphism' on *X*, and

iv) (given composable morphisms *f,g*), *c(f,g)* is the 'composite' of the pair *(f,g)*, and

v) the associativity and identity axioms are satisfied.

Non-canonical elements are [when C is a (small) category] the expressions

i) Ob(C) denoting the type of objects of C ;

ii) Mor(C,X,Y) denoting the type of morphisms of C from the object X to the object Y ;

iii) id denoting the identity morphism on any object ;

iv) comp(f,g) denoting the composite of two composable morphisms f,g .

Note that "i", "id" are polymorphic: the suffix, denoting an object, traditionally affixed to the symbol 1 to indicate dependence on the object, is unnecessary when the equations we handle are

between elements of a specified type [such as Mor(C,X,X)]. This polymorphism is traditional for the constant denoting composition, and may be extended to "id" without difficulty. As justification, note that the lambda-expression for the identity function makes no reference to the type on which the function acts.

We therefore now add the following rules :

CATform

$$\frac{}{\mathrm{CAT} : \mathrm{U1}}$$

CATintr

$$\frac{\begin{array}{l} O : U \\ M(X,Y) : U \; [\, X : O,\ Y : O \,] \\ i : M(X,X) \; [\, X : O \,] \\ c(f,g) : M(X,Z) \; [\, X : O,\ Y : O,\ Z : O,\ f : M(X,Y),\ g : M(Y,Z) \,] \\ c(f,i) = f : M(X,Y) \; [\, X : O,\ Y : O,\ f : M(X,Y) \,] \\ c(i,f) = f : M(X,Y) \; [\, X : O,\ Y : O,\ f : M(X,Y) \,] \\ c(f, c(g,h)) = c(c(f,g),h) : M(W,Z) \; [\, W : O,\ X : O,\ Y : O,\ Z : O, \\ \qquad f:M(W,X),\ g:M(X,Y),\ h:M(Y,Z) \,] \end{array}}{\mathrm{Cat}(O,M,i,c) : \mathrm{CAT}}$$

CATelim_Ob

$$\frac{C : \mathrm{CAT}}{\mathrm{Ob}(C) : U}$$

CATelim_Mor

$$\frac{X : \mathrm{Ob}(C) \quad Y : \mathrm{Ob}(C)}{\mathrm{Mor}(C,X,Y) : U}$$

CATelim_id

$$\frac{X : \mathrm{Ob}(C)}{\mathrm{id} : \mathrm{Mor}(C,X,X)}$$

CATelim_comp

$$\frac{f : \mathrm{Mor}(C,X,Y) \quad g : \mathrm{Mor}(C,Y,Z)}{\mathrm{comp}(f,g) : \mathrm{Mor}(C,X,Z)}$$

CATelim_IdL

$$\frac{f : \mathrm{Mor}(C,X,Y)}{\mathrm{comp}(f,\mathrm{id}) = f : \mathrm{Mor}(C,X,Y)}$$

CATelim_IdR

$$\frac{f : \mathrm{Mor}(C,X,Y)}{\mathrm{comp}(\mathrm{id},f) = f : \mathrm{Mor}(C,X,Y)}$$

CATelim_Ass

f : Mor(C,W,X) g : Mor(C,X,Y) h : Mor(C,Y,Z)

- -

comp(f,comp(g,h)) = comp(comp(f,g),h) : Mor(C,W,Z)

CATeq_Ob

Cat(O,M,i,c) : CAT

- -

Ob(Cat(O,M,i,c)) = O : U

CATeq_Mor

Cat(O,M,i,c) : CAT A : O B : O

- -

Mor(Cat(O,M,i,c),A,B) = M(A,B) : U

CATeq_id

Cat(O,M,i,c) : CAT A : O

- -

id = i : M(A,A)

CATeq_comp

Cat(O,M,i,c) : CAT
A:O
B:O
C:O
f : M(A,B)
g : M(B,C)

- - - - - - - - - - - - - - - - - - -

comp(f,g) = c(f,g) : M(A,C)

CATeq_eta

C:CAT

- -

C = Cat(Ob(C),Mor(C),id,comp) : CAT

This concludes our presentation of the formation, introduction, elimination and computation rules for the type "CAT" of (small) categories. Note that the computation rules (CATeq_Ob, CATeq_Mor, CATeq_id, CATeq_comp) are essential for relating the information wrapped up by the rule CATintr to that extracted from the category expression so introduced by the associated elimination rule.

5.2 Rules for functors

The following constants are added to the language :

"FUNC", "Func", "obj", "mor".

"FUNC" is a type constructor: when C, D are categories, we have the type "FUNC(C,D)" of functors from C to D. It has canonical elements of the form "Func(F1,F2)"; such an expression denotes the functor with object part F1, morphism part F2 [F1(X) being an Ob(C)-indexed family of objects of D, and F2(f) being a morphism: F1(X) → F1(Y) (when f is a morphism: X → Y), with F2 preserving identities and composites]. We note that FUNC(C,D) is, generally, a large type.

Moreover, when F is a functor from C to D, it has as objectpart the Ob(C)-indexed family obj(F,A) of objects of D, and asmorphism part the Mor(C,A,B)-indexed family mor(F,f) of morphisms of D from obj(F,A) to obj(F,B).

We thus have the following rules :

FUNCform

C : CAT D : CAT

FUNC(C,D) : U1

FUNCintr

C : CAT
F1(X) : Ob(D) [X : Ob(C)]
F2(f) : Mor(D,F1(X),F1(Y)) [X,Y : Ob(C), f : Mor(C,X,Y)]
F2(id) = id : Mor(D, F1(X), F1(X)) [X : Ob(C)]
F2(comp(f,g)) = comp(F2(f),F2(g)) : Mor(D,F1(X),F1(Z))
[X,Y,Z : Ob(C) , f : Mor(C,X,Y), g : Mor(C,Y,Z)]

Func(F1,F2) : FUNC(C,D)

FUNCelim_obj

F : FUNC(C,D) A : Ob(C)
---
obj(F,A) : Ob(D)

FUNCelim_mor

F : FUNC(C,D) f : Mor(C,A,B)
---
mor(F,f) : Mor(D,obj(F,A),obj(F,B))

FUNCelim_id

F : FUNC(C,D) A : Ob(C)
---
mor(F,id) = id : Mor(D,obj(F,A),obj(F,A))

FUNCelim_comp

F : FUNC(C,D) f : Mor(C,A,B) g : Mor(C,B,E)
---
mor(F,comp(f,g))=comp(mor(F,f),mor(F,g)): Mor(D,obj(F,A),obj(F,E))

FUNCeq_obj

Func(F1,F2) : FUNC(C,D) A : Ob(C)
---
obj(Func(F1,F2),A) = F1(A) : Ob(D)

FUNCeq_mor

Func(F1,F2) : FUNC(C,D) f : Mor(C,A,B)
---
mor(Func(F1,F2),f) = F2(f) : Mor(D,F1(A),F1(B))

FUNCeq_eta

F : FUNC(C,D)
- -
F = Func(obj(F), mor(F)) : FUNC(C,D)

This concludes our set of rules for the notion of functor.

5.3 Rules for natural transformations

The following constants are added to the language :

"NAT", "Nat", "at" .

"NAT(C,D,F,G)" denotes the type of natural transformations from F to G, with canonical elements of the form "Nat(n)", where "n(X)" is an expression denoting an Ob(C)-indexed family of morphisms from obj(F,X) to obj(G,X), satisfying a well-known equation. Moreover, if "nt" denotes a natural transformation from F to G, its component at an object A (of the domain category of F) is denoted by "at(nt,A)" .

Here now are the rules formalising these notions :

NATform

F : FUNC(C,D)   G : FUNC(C,D)
- -
NAT(C,D,F,G) : U1

NATintr

C:CAT
n(X) : Mor(D, obj(F,X),obj(G,X)) [X : Ob(C)]

comp(mor(F,f),n(Y))=comp(n(X),mor(G,f)) : Mor(D,obj(F,X),obj(G,Y))
[X,Y : Ob(C), f : Mor(C,X,Y)]
- -
Nat(n) : NAT(C,D,F,G)

NATelim_at

nt : NAT(C,D,F,G) A : Ob(C)
- -
at(nt,A) : Mor(D,obj(F,A),obj(G,A))

NATelim_comm

nt : NAT(C,D,F,G) f : Mor(C,A,B)
- -
comp(mor(F,f),at(nt,B)) = comp(at(nt,A),mor(G,f))
: Mor(D, obj(F,A), obj(G,B))

NATeq_at

Nat(n) : NAT(C,D,F,G) A : Ob(C)
- -
at(Nat(n),A) = n(A) : Mor(D,obj(F,A),obj(G,A))

NATeq_eta

n : Nat(C,D,F,G)
- -
n = Nat(at(n)) : Nat(C,D,F,G)

This concludes our system of rules for the notion of natural transformation.

5.4 Example : monomorphisms

As an example of the expressive power of the language, consider the notion of 'monomorphism'. We use of course the constructive definition "... if $g \cdot f = h \cdot f$ then $g = h$ ... " rather than the non-constructive "... if $g \neq h$ then $g \cdot f \neq h \cdot f$ ...". We add the definition

```
isMono(f,C,X,Y)
==
 Pi(Ob(C),(A)
   Pi(Mor(C,A,X), (g)
     Pi(Mor(C,A,X), (h)
       Á(Eq(Mor(C,A,Y), comp(g,f), comp(h,f)),
         Eq(Mor(C,A,X),g,h)))))
```

to the language, to be read 'f is a monomorphism of C from X to Y, when for every object A of C, and every morphism g of C from A to X, and every morphism h of C from A to X, g•f = h•f implies g = h.' This then allows us to construct a proof of the formula expressing the proposition that monomorphisms are closed under composition. Stripped of universal quantifiers, and thus assuming that C is a category, X,Y,Z are objects of C, f a morphism of C from X to Y, and f' a morphism of C from Y to Z, such a formula (preceded by a suitable proof-object) is as follows :

```
"lambda((hyp)[lambda((A)[lambda((g)[lambda((h)[lambda((ass)
   [apply(apply(apply(apply(fst(hyp),A),g),h),
   apply(apply(apply(apply(snd(hyp),A),comp(g,f)),comp(h,f)),e)
    )]])])])])]
:
  ->(#(isMono(f,C,X,Y),  isMono(f',C,Y,Z)),
  isMono(comp(f,f'),C,X,Z))

                [C:CAT; X:Ob(C); Y:Ob(C); f:Mor(C,X,Y); Z:Ob(C); f':Mor(C,Y,Z)]"
```

Construction of this proof is routine, but tedious. The associativity axiom is used at one point, but its use is no longer visible, having been absorbed into the unique canonical proof object *"e"* of the formula *"Eq(Mor(C,A,X),g,h))"*. Similar definitions and proofs for other basic notions, such as epimorphisms, can clearly be given.

5.5 Example : the category of categories

Functors are composable, as in [24] p14: the (polymorphic) identity functor *"Id"* may be defined as *"Func((X)X, (f)f)"*, and routinely shown to be of type *"FUNC(C,C)"* when *C* is a category , and the composite *"FF_comp(F,G)"* of two functors *F : FUNC(C,D), G : FUNC(D,E)* defined to be

Func((X)obj(G,obj(F,X)), (f)mor(G,mor(F,f))) .

Likewise, the vertical composite ([24], p.40) of two natural transformations $m : F \to G,\ n : G \to H$ can be defined as

NN_vert(m,n) == Nat((X)comp(at(m,X), at(n,X)))

and for this composition there is the (polymorphic) identity natural transformation "Id_nat" defined as "Nat((x)id)". There is also the composite of a functor *F* with a natural transformation *n* by

FN_comp(F,n) == Nat((X)at(n,obj(F,X)))

and, similarly, the composition NF_comp(n,F). But, the horizontal composition is more difficult: there are two ways to define it, for m : NAT(C,D,S,S'), n : NAT(D,E,T,T'); these are provably equivalent ([24], p43):

NN_horiz1(m,n,S,S',T,T') == NN_vert(NF_comp(m,T),FN_comp(S',n))
NN_horiz2(m,n,S,S',T,T') == NN_vert(FN_comp(S,n),NF_comp(m,T'))

Here, we pay the price for not having an operation 'Dom' so that "Dom(n)" is the domain of the natural transformation n, and thus we have to abstract over the names of at least two of the functors involved. (We do it in both cases over all four, to be consistent). Moreover, forcing the GTTS to be clever enough to recognise these two definitions as the same is a major problem, symptomatic of other, similar, difficulties.

6. Comparison with axiomatic approach

Two rather different approaches are possible to the representation of a special type, such as CAT, in the type theory system. One is that adopted above, where we add new constants to the language, and new rules to manipulate these constants. The other is not to extend the language (save perhaps with some inessential definitions to make life more convenient), but to derive a judgment that such and such an expression denotes a type. For example, here is one possible representation of the abstract type "CAT", which has been shown, by manual interaction with the type theory system, to denote a type.

```
"Sigma(U, (O)
    [Sigma(->(#(O,O),U), (M)
        [Sigma(Pi(O,(o)[apply(M,pair(o,o))]), (ident)
            [Sigma(Pi (O, (o1) [Pi (O, (o2) [Pi (O, (o3)
                [-> (# (apply(M,pair(o1,o2)), apply(M,pair(o2,o3))),
                    apply(M,pair(o1,o3)))])])]), (com)
                    [#(#(Pi(O,(o1)[Pi(O,(o2)
                        [Pi(apply(M,pair(o1,o2)),(f)
                            [Eq(apply(M,pair(o1,o2)),
                                apply(apply(apply(apply(com,o1),o1),o2),
                                    pair(apply(ident,o1),f)),
                                f)])])]),
 Pi(O,(o1)[Pi(O,(o2)[Pi(apply(M,pair(o1,o2)),(f)
 [Eq(apply(M,pair(o1,o2)),
      apply(apply(apply(apply(com,o1),o2),o2),
           pair(f,apply(ident,o2))),
     f)])])])),
 Pi(O,(o1)[Pi(O,(o2)[Pi(O,(o3)[Pi(O,(o4)
 [Pi(apply(M,pair(o1,o2)), (f)
 [Pi(apply(M,pair(o2,o3)), (g)
 [Pi(apply(M,pair(o3,o4)), (h)
 [Eq(apply(M,pair(o1,o4)),
  apply(apply(apply(apply(com,o1),o3),o4),
  pair(apply(apply(apply(apply(com,o1),o2),o3),
 pair(f,g)),h)),
  apply(apply(apply(apply(com,o1),o2),o4),
  pair(f,apply(apply(apply(apply(com,o2),o3),o4),
 pair(g, h)))))
 ])])])])])])]))])])])"
```

This follows the approach advocated in Nordström & Petersson [29] for representing abstract data types, such as stacks. Several serious problems were found with this approach. The first

problem encountered arose from the lack, in the GTTS, of primitive rules for extracting from an expression of a Sigma-type the two components. This omission is logically acceptable, but inconvenient: in order to extract such components, one has to use the primitive rule SIGMAelim, and in order to do that, one has to know the types of the components. Appropriate new primitive rules, such as SIGMAelim_fst, have in fact been added [14] to the version of the system which we are now using. The second difficulty is even more awkward: one only has the notation "fst(C)" for the type of objects of the category C, "fst(snd(C))" for the type of its morphisms, ... Any attempt to coerce the GTTS into saying "Ob(C)", "Mor(C)", etc. will have undesirable consequences elsewhere. Third, the notation is clumsy: note the many uses of "apply" in the above type-expression. We are indebted to Lincoln Wallen for encouraging the abandonment of this approach, and the adoption of ideas advocated in [39].

Such disdain for use of primitive concepts, however, forces the construction of more theorem-proving software, for handling the new constants. Since the nature of such software is dictated in part by the methods used by category theorists for proving their theorems, rather than by the general methods used elsewhere, this may yet turn out to be an advantage.

The NuPRL system [09] encourages the representation of categories in axiomatic form: the corresponding NuPRL type expression is as follows:

```
Category ==
 O : U1
 # M : ( O # O ) -> U1
 # id : (o : O -> M(o,o))
 # comp :( o1 : O -> o2 : O -> o3 : O ->
        ( M(o1,o2)  #  M(o2,o3)  ->  M(o1,o3)))
 # idL : All o1 : O . All o2 : O . All f : M(o1,o2) .
        comp(o1)(o1)(o2)(id(o1),f)  =  f  in  M(o1,o2)
 # idR : All o1 : O . All o2 : O . All f : M(o1,o2) .
        comp(o1)(o2)(o2)(f,id(o2))  =  f  in  M(o1,o2)
 # Ass : All o1 : O . All o2 : O . All o3 : O . All o4 : O.
        All f : M(o1,o2). All g : M(o2,o3) . All h : M(o3,o4).
        comp(o1)(o3)(o4)(comp(o1)(o2)(o3)(f,g),  h)  =
        comp(o1)(o2)(o4)(f,comp(o2)(o3)(o4)(g,h))  in  M(o1,o4)
```

which is less clumsy, but suffers from the second (and to some extent, the third) problem mentioned above.

7. Automated categorial logic

Representations of the basic concepts of category theory, and of rules adequate in principle for the derivation of theorems thereof, are but steps towards an automated theorem-proving system. In developing such a system on top of the present representation, there are many tasks to be mechanised: for example -

i) implementation of a top-down proof editor;

ii) derivation of routine 'equational' theorems of category theory;

iii) derivation of hypothetical and non-equational theorems of category theory;

iv) extension of such methods to, for example, the theory of cartesian closed categories;

iv) derivation of theorems about functors and natural transformations.

A proof-editor for the unmodified GTTS has been implemented recently by Hamilton [19], and we intend to incorporate this into our system. Rewriting techniques developed by Curien [12] are adequate for the solution of word problems in the theory of categories, the theory of categories with binary products, and (to some extent) the theory of cartesian-closed categories. Word problems in the latter theory are also solvable [23] by conversion to lambda-calculus, and (in a paper [30] not yet available to me) by a decision procedure due to Obtulowicz. There are substantial problems in using ordinary rewriting techniques for all but the simplest problems: termination proofs are difficult, and require techniques from proof theory (as noted by Huet [22]). Ordinary equational reasoning in category theory depends heavily on the associativity of composition: it would be natural to build this [34] into the underlying unification algorithm, if rewriting techniques are to be used, except that minimal complete sets of unifiers may be infinite. The associative axiom can be expressed as a rewrite rule, but this appears to force the Knuth-Bendix completion algorithm, as implemented in REVE 2.4 [15], (when fed, for example, with equational axioms for cartesian-closed categories) to generate new rules indefinitely. Thus, much work in this area remains to be done.

8. References

[01] Barr, M. & Wells, C. : *Toposes, triples, and theories*; Grundlehren der mathematischen Wissenschaften **278**, Springer-Verlag 1985 .

[02] Beeson, M. : *Foundations of Constructive Mathematics*; Ergebnisse der Mathematik und ihrer Grenzgebiete, **3** Folge, Band **6**, Springer-Verlag 1985.

[03] Benecke, K. & Reichel, H. : *Equational partiality*; Algebra Universalis, **16** (1983), pp 219-232.

[04] Burmeister, P. : *Partial algebras - survey of a unifying approach towards a two-valued model theory for partial algebras*; Algebra Universalis **15** (1982), pp 306-358 .

[05] Burstall, R.M. : *Electronic category theory* ; Proc. Math. Found. C. Sci., ed Dembinski, LNCS **88**, Springer-Verlag 1980, pp 22-39.

[06] Burstall, R.M. & Rydeheard, D.E. : Computational category theory; Prentice-Hall, 1988.

[07] Cartmell, J. : *Generalised algebraic theories and contextual categories*; D.Phil thesis, University of Oxford, (1978) .

[08] Cartmell, J. : *Reduction rules for cartesian closed categories*; (typescript, University of Edinburgh, 1985).

[09] Constable, R. et al. : *Implementing mathematics with the NuPRL proof development system* ; Prentice-Hall, 1986.

[10] Coquand, T. & Huet, G. : *A theory of constructions*; preprint (Data Types Symposium, Sophia-Antipolis), (1984).

[11] Coquand, T. & Huet, G. : *Constructions : a higher order proof system for mechanising mathematics*; preprint (EUROCAL 85, Linz), (1985).

[12] Curien, P. - L. : *Categorical combinators, sequential algorithms and functional programming*; (thèse d'état) (CNRS - Université Paris VII, LITP, No 85-26) (March 1985).

[13] Dyckhoff, R. : *Mechanical diagram chasing;* (Univ. Edinburgh M.Sc. thesis, September 1983).

[14] Dyckhoff, R. : *Derived and meta-derived rules in Martin-Löf type theory*; preprint, St Andrews, June 1985.

[15] Forgaard, R. & Guttag, J.V. : *REVE: a term rewriting system generator with failure-resistant Knuth-Bendix,* MIT 1984.

[16] Freyd, P. : *Aspects of topoi*; Bull. Austral. Math. Soc. **7** (1972), pp 1-76 and 467-480 .

[17] Gordon, M.J., Milner, R. & Wadsworth, C.P. : *Edinburgh LCF*; Lecture Notes in Computer Science **78** , Springer-Verlag 1979.

[18] Gray, J. : *Categorical aspects of parametric data types*, (preprint, University of Illinois, Urbana-Champaign, May 1985).

[19] Hamilton, A. : *Program construction in Martin-Löf type theory* ; Tech. Rept. **24**, University of Stirling, June 1985.

[20] Hamza, T.T.A. : *Normalisation techniques in proof theory and category theory - an implementation and applications*; Ph. D. thesis, St Andrews, 1985.

[21] Herrlich, H. & Strecker, G. : *Category Theory* ; Sigma Series in Pure Mathematics **1**, Heldermann-Verlag, Berlin 1979.

[22] Huet, G. : *Equational systems for category theory and intuitionistic logic*; talk at Rewriting Techniques & Applications Conference, Dijon, May 1985.

[23] Lambek, J. & Scott, P. : *Introduction to higher order categorical logic*; (Cambridge U.P., 1986)

[24] MacLane, S. : *Categories for the working mathematician*; Graduate Texts in Mathematics **5**, Springer-Verlag 1971.

[25] Martin-Löf, P. : *An intuitionistic theory of types : predicative part*; in: Rose & Shepherdson, Eds., Logic Colloquium **73**, (North-Holland, Amsterdam, 1975) pp 73-118.

[26] Martin-Löf, P. : *Constructive mathematics and computer programming* ; in: Logic, Methodology and Philosophy of Science IV, (North-Holland, Amsterdam, 1979) pp 153-175.

[27] Martin-Löf, P. : *Intuitionistic type theory*; Bibliopolis, 1984.

[28] Nordström, B. & Petersson, K. : *Types and specifications*; in : Info. Processing **83**, ed. R.E.A.Mason, North-Holland 1983, pp 915-920.

[29] Nordström, B. & Petersson, K. : *The semantics of module specifications in Martin-Löf's type theory*, (draft, Göteborg, Jan 1985).

[30] Obtulowicz, A. : *Algebra of Constructions I. The word problem for partial algebras*, submitted to Information & Control (1985).

[31] Petersson, K. : *A programming system for type theory* ; LPM memo **21**, Department of Computer Science, Chalmers University of Technology, Göteborg (1982, 1984).

[32] Petersson, K. : *The subset type former and the type of small types in Martin-Löf's theory of types*; LPM memo **33**, Department of Computer Science, Chalmers University of Technology, Göteborg (1984).

[33] Pitt, D. (editor) : *Category theory and computer programming*, Lecture Notes in Computer Science **240**, Springer-Verlag 1986.

[34] Plotkin, G. : *Building in equational theories*; Machine Intelligence **7** (1972).

[35] Poigné, A. : *On specifications, theories and models with higher types*; preprint, Imperial College London 1984 .

[36] Poigné, A. & Voss, J. : *Programs over abstract data types - on the implementation of abstract data types*; (preprint) Dortmund 1984.

[37] Rydeheard, D.E. : *Applications of category theory to programming and program specification*; Univ. of Edinburgh Ph.D. thesis, 1981

[38] Rydeheard, D.E. & Burstall, R.M. : *The unification of terms, a category- theoretic algorithm* ; (preprint, May 1985).

[39] Schmidt, D. : *Natural deduction theorem-proving in set theory,* internal report CSR-142-83, Edinburgh Univ. C.S. Dept, 1983.

[40] Scott, D. : *Identity and existence in intuitionistic logic* ; Applications of Sheaves, Lecture Notes in Mathematics **753**, Springer-Verlag 1979, pp 660-696 .

[41] Seely, R.A.G. : *Locally cartesian closed categories and type theory*; Math. Proc. Camb. Phil. Soc. **95** (1984), pp 33-48.

[42] Szabo, M. : *Algebra of proofs*; North-Holland (1978).

[43] Watjen, D. & Struckmann, W. : *An algorithm for verifying equations of morphisms in a category; Information processing letters* **14** (1982), pp 104-108.