Automating Proofs in Category Theory

Dexter Kozen[1], Christoph Kreitz[1,2], and Eva Richter[2]

[1] Computer Science Department,
Cornell University, Ithaca, NY 14853-7501, USA
[2] Institut für Informatik,
Universität Potsdam, 14482 Potsdam, Germany

Abstract. We introduce a semi-automated proof system for basic category-theoretic reasoning. It is based on a first-order sequent calculus that captures the basic properties of categories, functors and natural transformations as well as a small set of proof tactics that automate proof search in this calculus. We demonstrate our approach by automating the proof that the functor categories $\mathsf{Fun}[\mathsf{C} \times \mathsf{D}, \mathsf{E}]$ and $\mathsf{Fun}[\mathsf{C}, \mathsf{Fun}[\mathsf{D}, \mathsf{E}]]$ are naturally isomorphic.

1 Introduction

Category theory is a popular framework for expressing abstract properties of mathematical structures. Since its invention in 1945 by Samuel Eilenberg and Saunders Mac Lane [12], it has had a wide impact in many areas of mathematics and computer science. The beauty of category theory is that it allows one to be completely precise about general mathematical concepts. Abstract algebraic notions such as free constructions, universality, naturality, adjointness, and duality have precise formulations in the theory. Many algebraic constructions become exceedingly elegant at this level of abstraction.

However, there are some disadvantages too. Many basic facts, although easy to state, can be quite tedious to verify formally. Diagrams can be used to illustrate essential insights, but complete proofs based on precise definitions often involve an enormous number of low-level details that must be checked. In many cases, it is not considered worth the effort to carry out such a detailed verification, and readers are frequently asked to accept "obvious" assertions on faith.

Another issue is that category theory is considerably more abstract than many other branches of mathematics. Because of this abstraction, it is easy to lose sight of the connection with concrete motivating examples. One works in a rarified atmosphere in which much of the intuition has been stripped away, so the verification at the lowest level becomes a matter of pure symbol manipulation, devoid of motivating intuition.

On the other hand, precise proofs in category theory often rely on standard patterns of reasoning that may lend themselves well to automation. Providing such an automation serves two purposes. It enables users to generate completely formal proofs of elementary category-theoretic facts without having to go through all the details themselves, thus providing assurance that the statement is in fact true and allowing them to inspect details if desired. It also demonstrates that the proofs that many authors do not bother to provide, which may be considered trivial from an intellectual point of view, actually may contain a tremendous amount of hidden detail, and may identify conditions that formally should be checked, but that the author might have taken for granted or overlooked entirely.

In this paper we introduce a proof system for automating basic category-theoretic reasoning. We first give a formal first-order axiomatization of elementary category theory that is amenable to automation in Section 2. This axiomatization is a slight modification of a system presented in [18]. We then describe an implementation of this calculus within the proof environment of the Nuprl system [9,1] in Section 3 and strategies for automated proof search in Section 4. These strategies attempt to capture the general patterns of formal reasoning that we have observed in hand-constructed proofs using this calculus. These patterns were alluded to in [18], but the description there was quite vague and there was no attempt at implementation.

We demonstrate the feasibility of our approach by giving a completely automated proof of the statement that the functor categories $\mathsf{Fun[C \times D, E]}$ and $\mathsf{Fun[C, Fun[D, E]]}$ are naturally isomorphic. The process of automating this proof has given us significant insights into the formal structure of category-theoretic proofs and has taught us much about how to streamline the automation. We describe these technical insights below in the context of the proof itself.

1.1 Related Work

The published approaches to a formalization of category theory essentially aim at three different purposes. The first is a formal reconstruction of mathematical knowledge in a computer-oriented environment. This is done in the Mizar project of Bialystok University [29]. Mizar statements are formulated in first order logic and proved using a declarative proof language. Mizar's library contains a comprehensive collection of theorems mostly proved already in 1990-1996, but is still under active research. The last entries concerning special functor behavior and duality of categories were done in 2001 [3,4,5]. One disadvantage of the Mizar approach is that it has only little automation: although Mizar's basic inference steps are quite expressive, it does not provide a mechanism for automating domain-specific reasoning tasks.

A second purpose is to provide calculi for category theory to use its machinery in several domains of computer science (for example denotational semantics). One of these approaches is Caccamo's and Winskel's *Higher order calculus for categories* [10]. The authors present a second order calculus for a fragment of category theory. Their approach is at a level higher than ours. Their basic types are (small) categories and the syntactic judgments describe functorial behavior of expressions. The rules allow the construction of new functors. A consequence of this approach is that for example Yoneda's lemma occurs as a rule rather than a theorem. Another approach to be mentioned here is Rydeheard's and Burstall's *Computational Category Theory* [24]. This work is a programming language representation of the subject, i.e., a collection of structures and functions that represent the main concepts of category theory. One of the merits of the book is that it emphasizes the constructive flavor of categorial concepts. The representation is mainly regarded a basis for the use of category theory in program design.

A third group consists of formalizations of category theory in interactive proof systems. In these formalizations, practical issues like feasibility, automation and elegance of the design (in the sense of [15]) play an important role. There are at least two formalizations of category theory in Isabelle/HOL that should be mentioned here. Glimming's 2001 master thesis [13] describes a development of basic category theory and a couple of concrete categories. As HOL does not admit the definition of partial functions, Glimming had to address the problem of the composition of uncomposable arrows. This problem is solved by the introduction of an error object, which is never a member of any set of arrows. Since his interests lie in a formalization of the Bird-Meertens formalism [7], there are no attempts to improve automation beyond Isabelle's generic prover.

Another formalization of category theory in Isabelle is O'Keefe's work described in [22]. His main focus is on the readability of the proofs, aiming at a representation close to one in a mathematical textbook. Therefore he uses a sectioning concept provided by Isabelle. This saves a lot of repetition and is an elegant way to emulate informal mathematical reasoning. Although this formalization contains definitions of functors and natural transformations, it does not include functor categories. O'Keefe mentions another formalization of category theory in HOL by Lockwood Morris whose focus is on automation, but unfortunately neither a description nor the sources have been published.

In the Coq library there are two contributions concerning category theory. The development of Saïbi and Huet [16,26] contains definitions and constructions up to cartesian closed categories, which are then applied to the category of sets. The authors formulate the theory of functors including Freyd's adjoint functor theorem, i.e., their work covers nearly all of chapters I–V of [20]. The formalization of Saïbi and Huet is directly based on the constructive type theory of Coq. Simpson [27], on the other hand, makes only indirect use of it. Instead, his formalization is set up in a ZFC-like environment. In addition to some basic set theory and algebra, he develops category theory including functors, natural transformations, limits and colimits, functor categories, and a theorem about the existence of (co)limits in functor categories. Simpson has written some tactics to improve the automation, but, as for the work of Saïbi and Huet, there are no official papers available.

A key difference between these works and our approach is that we have identified an independent calculus for reasoning about category theory and given a full implementation in Nuprl. In addition, we have provided a family of tactics that allow many proofs to be automated. None of the other extant implementations we have encountered make any attempt to isolate an independent formal axiomatization of the elementary theory. Instead, they embed category theory into some other logic, and reasoning relies mostly on the underlying logic.

2 An Axiomatization of Elementary Category Theory

2.1 Notational Conventions

We assume familiarity with the basic definitions and notation of category theory [6,20]. To simplify notation, we will adhere to the following conventions.

- Symbols in sans serif, such as C, always denote categories. The categories Set and Cat are the categories of sets and set functions and of (small) categories and functors, respectively.
- If C is a category, we use the symbol C to denote both the category C and the set of objects of C.
- We write $A : \mathsf{C}$ to indicate that A is an object of C. Composition is denoted by the symbol $\circ$ and the identity on object $A : \mathsf{C}$ is denoted 1_A. The use of a symbol in sans serif, such as C, implicitly carries the type assertion $\mathsf{C} : \mathsf{Cat}$.
- We write $h : \mathsf{C}(A, B)$ to indicate that h is an arrow of the category C with domain A and codomain B.
- $\mathsf{Fun}\,[\mathsf{C}, \mathsf{D}]$ denotes the functor category whose objects are functors from C to D and whose arrows are natural transformations on such functors. This is the same as the category denoted D^{C} in [20]. Thus $F : \mathsf{Fun}\,[\mathsf{C}, \mathsf{D}]$ indicates that F is a functor from C to D and $\varphi : \mathsf{Fun}\,[\mathsf{C}, \mathsf{D}]\,(F, G)$ indicates that φ is a natural transformation with domain F and codomain G.
- C^{op} denotes the opposite category of C.
- $f : X \Rightarrow Y$ indicates that $f : \mathsf{Set}(X, Y)$, that is, f is a set function from set X to set Y. We use the symbol $\Rightarrow$ only in this context. Function application is written as juxtaposition and associates to the left.
- F^1 and F^2 denote the object and arrow components, respectively, of a functor F. Thus if $F : \mathsf{Fun}\,[\mathsf{C}, \mathsf{D}]$, $A, B : \mathsf{C}$, and $h : \mathsf{C}(A, B)$, then $F^1A, F^1B : \mathsf{D}$ and $F^2h : \mathsf{D}(F^1A, F^1B)$.
- Function application binds tighter than the operators ^{1} and ^{2}. Thus the expression F^1A^2 should be parsed $(F^1A)^2$.
- $\mathsf{C} \times \mathsf{D}$ denotes the product of categories C and D. Its objects are pairs $(A, X) : \mathsf{C} \times \mathsf{D}$, where $A : \mathsf{C}$ and $X : \mathsf{D}$, and its arrows are pairs $(f, h) : (\mathsf{C} \times \mathsf{D})((A, X), (B, Y))$, where $f : \mathsf{C}(A, B)$ and $h : \mathsf{D}(X, Y)$. Composition and identities are defined componentwise; that is,

$$(g, k) \circ (f, h) \overset{\text{def}}{=} (g \circ f, k \circ h) \tag{1}$$

$$1_{(A,X)} \overset{\text{def}}{=} (1_A, 1_X). \tag{2}$$

2.2 Rules

The rules involve sequents $\Gamma \vdash \alpha$, where Γ is a type environment (set of type judgments on atomic symbols) and α is either a type judgment or an equation. There is a set of rules for functors and a set for natural transformations, as well as some rules covering the basic properties of categories and equational reasoning.

The rules for functors and natural transformations are the most interesting. They are divided into symmetric sets of rules for analysis (elimination) and synthesis (introduction).

Categories. There is a collection of rules covering the basic properties of categories, which are essentially the rules of typed monoids. These rules include typing rules for composition and identities

$$\frac{\Gamma \vdash A, B, C : \mathsf{C}, \quad \Gamma \vdash f : \mathsf{C}(A, B), \quad \Gamma \vdash g : \mathsf{C}(B, C)}{\Gamma \vdash g \circ f : \mathsf{C}(A, C)} \tag{3}$$

$$\frac{\Gamma \vdash A : \mathsf{C}}{\Gamma \vdash 1_A : \mathsf{C}(A, A)}, \tag{4}$$

as well as equational rules for associativity and two-sided identity.

Functors. A functor F from C to D is determined by its object and arrow components F^1 and F^2. The components must be of the correct type and must preserve composition and identities. These properties are captured in the following rules.

Analysis

$$\frac{\Gamma \vdash F : \mathsf{Fun}\,[\mathsf{C}, \mathsf{D}], \quad \Gamma \vdash A : \mathsf{C}}{\Gamma \vdash F^1 A : \mathsf{D}} \tag{5}$$

$$\frac{\Gamma \vdash F : \mathsf{Fun}\,[\mathsf{C}, \mathsf{D}], \quad \Gamma \vdash A, B : \mathsf{C}, \quad \Gamma \vdash f : \mathsf{C}(A, B)}{\Gamma \vdash F^2 f : \mathsf{D}(F^1 A, F^1 B)} \tag{6}$$

$$\frac{\Gamma \vdash F : \mathsf{Fun}\,[\mathsf{C}, \mathsf{D}], \quad \Gamma \vdash A, B, C : \mathsf{C}, \quad \Gamma \vdash f : \mathsf{C}(A, B), \quad \Gamma \vdash g : \mathsf{C}(B, C)}{\Gamma \vdash F^2(g \circ f) = F^2 g \circ F^2 f} \tag{7}$$

$$\frac{\Gamma \vdash F : \mathsf{Fun}\,[\mathsf{C}, \mathsf{D}], \quad \Gamma \vdash A : \mathsf{C}}{\Gamma \vdash F^2 1_A = 1_{F^1 A}} \tag{8}$$

Synthesis

$$\frac{\begin{array}{l} \Gamma,\ A : \mathsf{C} \vdash F^1 A : \mathsf{D} \\ \Gamma,\ A, B : \mathsf{C},\ g : \mathsf{C}(A, B) \vdash F^2 g : \mathsf{D}(F^1 A, F^1 B) \\ \Gamma,\ A, B, C : \mathsf{C},\ f : \mathsf{C}(A, B),\ g : \mathsf{C}(B, C) \vdash F^2(g \circ f) = F^2 g \circ F^2 f \\ \Gamma,\ A : \mathsf{C} \vdash F^2 1_A = 1_{F^1 A} \end{array}}{\Gamma \vdash F : \mathsf{Fun}\,[\mathsf{C}, \mathsf{D}]} \tag{9}$$

Natural Transformations. A natural transformation $\varphi : \mathsf{Fun}\,[\mathsf{C}, \mathsf{D}]\,(F, G)$ is a function that for each object $A : \mathsf{C}$ gives an arrow $\varphi A : \mathsf{D}(F^1 A, G^1 A)$, called the *component* of φ at A, such that for all arrows $g : \mathsf{C}(A, B)$, the following diagram commutes:

$$\begin{tikzcd} F^1 A \arrow[r, "F^2 g"] \arrow[d, "\varphi A"'] & F^1 B \arrow[d, "\varphi B"] \\ G^1 A \arrow[r, "G^2 g"] & G^1 B \end{tikzcd} \tag{10}$$

Composition and identities are defined by

$$(\varphi \circ \psi) A \overset{\text{def}}{=} \varphi A \circ \psi A \tag{11}$$

$$1_F A \overset{\text{def}}{=} 1_{F^1 A}. \tag{12}$$

The property (10), along with the typing of φ, are captured in the following rules.

Analysis

$$\frac{\Gamma \vdash \varphi : \mathsf{Fun}\,[\mathsf{C},\mathsf{D}]\,(F,G)}{\Gamma \vdash F, G : \mathsf{Fun}\,[\mathsf{C},\mathsf{D}]} \tag{13}$$

$$\frac{\Gamma \vdash \varphi : \mathsf{Fun}\,[\mathsf{C},\mathsf{D}]\,(F,G), \quad \Gamma \vdash A : \mathsf{C}}{\Gamma \vdash \varphi A : \mathsf{D}(F^1 A, G^1 A)} \tag{14}$$

$$\frac{\Gamma \vdash \varphi : \mathsf{Fun}\,[\mathsf{C},\mathsf{D}]\,(F,G), \quad \Gamma \vdash A, B : \mathsf{C}, \quad \Gamma \vdash g : \mathsf{C}(A,B)}{\Gamma \vdash \varphi B \circ F^2 g = G^2 g \circ \varphi A} \tag{15}$$

Synthesis

$$\frac{\begin{array}{l}\Gamma \vdash F, G : \mathsf{Fun}\,[\mathsf{C},\mathsf{D}] \\ \Gamma,\ A : \mathsf{C} \vdash \varphi A : \mathsf{D}(F^1 A, G^1 A) \\ \Gamma,\ A, B : \mathsf{C},\ g : \mathsf{C}(A,B) \vdash \varphi B \circ F^2 g = G^2 g \circ \varphi A\end{array}}{\Gamma \vdash \varphi : \mathsf{Fun}\,[\mathsf{C},\mathsf{D}]\,(F,G)} \tag{16}$$

Equational Reasoning. Besides the usual domain-independent axioms of typed equational logic (reflexivity, symmetry, transitivity, and congruence), certain domain-dependent equations on objects and arrows are assumed as axioms, including the associativity of composition and two-sided identity rules for arrows, the equations (1) and (2) for products, and the equations (11) and (12) for natural transformations.

We also provide extensionality rules for objects of functional type:

$$\frac{\Gamma \vdash F, G : \mathsf{Fun}\,[\mathsf{C},\mathsf{D}], \quad \Gamma, A : \mathsf{C} \vdash F^1 A = G^1 A}{\Gamma \vdash F^1 = G^1} \tag{17}$$

$$\frac{\Gamma \vdash F, G : \mathsf{Fun}\,[\mathsf{C},\mathsf{D}], \quad \Gamma,\ A, B : \mathsf{C},\ g : \mathsf{C}(A,B) \vdash F^2 g = G^2 g}{\Gamma \vdash F^2 = G^2} \tag{18}$$

$$\frac{\Gamma \vdash F, G : \mathsf{Fun}\,[\mathsf{C},\mathsf{D}], \quad \Gamma \vdash F^1 = G^1, \quad \Gamma \vdash F^2 = G^2}{\Gamma \vdash F = G} \tag{19}$$

$$\frac{\Gamma \vdash F, G : \mathsf{Fun}\,[\mathsf{C},\mathsf{D}], \quad \Gamma \vdash \varphi, \psi : \mathsf{Fun}\,[\mathsf{C},\mathsf{D}]\,(F,G), \quad \Gamma,\ A : \mathsf{C} \vdash \varphi A = \psi A}{\Gamma \vdash \varphi = \psi} \tag{20}$$

Finally, we also allow equations on types and substitution of equals for equals in type expressions. Any such equation $\alpha = \beta$ takes the form of a rule

$$\frac{\Gamma \vdash A : \alpha}{\Gamma \vdash A : \beta}. \tag{21}$$

Other Rules. There are also various rules for products, weakening, and other structural rules for manipulation of sequents. These are all quite standard and do not bear explicit mention.

3 Implementation of the Formal Theory

As a platform for the implementation of our proof calculus we have selected the Nuprl proof development system [9,2,19,1]. Nuprl is an environment for the development of formalized mathematical knowledge that supports interactive and tactic-based reasoning, decision procedures, language extensions through user-defined concepts, and an extendable library of verified formal knowledge. Most of the formal theories in this library are based on Nuprl's Computational Type Theory, but the system can accommodate other logics as well.

One of the key aspects of the implementation of a formal theory is faithfulness with respect to the version on paper. Although Nuprl supports a much more expressive formalism, reasoning mechanisms should be restricted to first-order logic and the axiomatization of category theory given in the previous section. To accomplish this we proceeded as follows.

Encoding Semantics and Syntax. We have used Nuprl's definition mechanism to implement the vocabulary of basic category theory. For each concept we have added an abstraction object to the library that defines the semantics of a new abstract term in terms of Nuprl's Computational Type Theory. For instance, the product of two categories C and D, each consisting of a set of objects, a set of arrows, a domain and a codomain function, and composition and identity operations, is defined by an abstraction which states that objects and arrows are the cartesian products of the respective sets for C and D, domain and codomain functions are paired, and composition and identity are computed pointwise.

```
CatProd(C,D) == < C×D,  ArrC×ArrD,  λ(f,h).<dom(f),dom(h)>,
      λ(f,h).<cod(f),cod(h)>, λ((f,h),(g,k)).<f∘g,h∘k>, λ(A,X).<1A,1X> >
```

The outer appearance of abstract terms (display syntax) is defined separately through *display forms*, which enable us to adjust the print and display representations of abstract terms to conform to a specific style without modifying the term itself. Following [20], for instance, we denote the set of objects of a category by the name of the category. For the product category, we use the same notation as for the cartesian product.

```
C×D == CatProd(C,D)
```

Since the abstract terms are different, the proof system can easily distinguish terms that look alike but have a different meaning. Display forms can also be used to suppress information that is to be considered implicit. The composition of two arrows f and g, for instance, depends on the category C to which f and g belong, but it would be awkward to write down this information every time the composition operator is used.

Currently, Nuprl's display is restricted to a single 8-bit font. This limits the use of symbols, subscripts and superscripts to the characters defined in this font. Identities, usually written as 1_A or $1_{(A,X)}$, have to be presented as `1A` and `1<A,X>`. Apart from these restrictions, all the basic category-theoretic vocabulary appears in the system as described in Section 2.

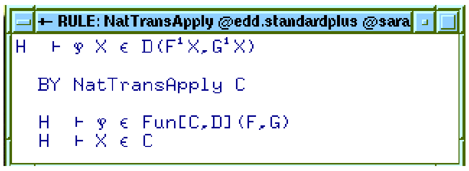
Inference rules. Given the formal representation of basic category theory, there are several ways to implement the rules.

The standard approach would be to encode rules as tactics based on elementary inference rules. However, it is difficult to prove that these tactics actually represent a specific category-theoretic rule. Furthermore, the tactics may require executing hundreds of basic inferences for each category-theoretic inference step. A more efficient way is to write tactics based on formal theorems that establish properties of the fundamental concepts. For instance, rule (14) corresponds to the theorem

$$\forall\texttt{C,D:Categories.}\ \forall\texttt{F,G:Fun[C,D].}\ \forall\varphi\texttt{:Fun[C,D](F,G).}\ \forall\texttt{X:C.}\ \varphi\texttt{X} \in \texttt{D(F}^1\texttt{X,G}^1\texttt{X)}$$

To apply the rule, one would instantiate the theorem accordingly. But this would lead to proof obligations that do not occur in the original rule, such as showing that C and D are categories and F and G are functors in Fun[C, D].

The Nuprl system supports a more direct approach to encoding formal theories. Experienced users can add rule objects to the system's library that directly represent the inference rules of the theory, then prove formal theorems like the one above to justify the rules. Apart from the fact that rules have to be formulated as top-down sequent rules to accommodate Nuprl's goal-oriented reasoning style, the representation of the rules in the system is identical to the version on paper, which makes it easy to check its faithfulness. Rule (14), for instance, is represented by a rule object `NatTransApply` with the following contents.



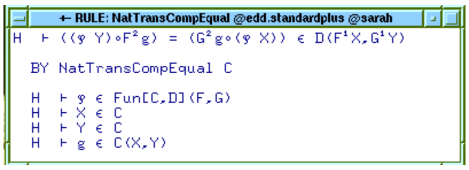
```
H  ⊢ φ X ∈ D(F^1X,G^1X)

   BY NatTransApply C

   H  ⊢ φ ∈ Fun[C,D](F,G)
   H  ⊢ X ∈ C
```

Nuprl's rule compiler converts rule objects into rules that match the first line of the object against the actual goal sequent and create the subgoal sequents by instantiating the two lower lines. Note that the rule requires the category C to be given as parameter, since it occurs in a subgoal but not in the main goal.

Since equalities in Nuprl are typed, we added types to all the inference rules that deal with equalities. For example, rule (15) is represented as follows:



```
H  ⊢ ((φ Y)∘F^2g) = (G^2g∘(φ X)) ∈ D(F^1X,G^1Y)

  BY NatTransCompEqual C

  H  ⊢ φ ∈ Fun[C,D](F,G)
  H  ⊢ X ∈ C
  H  ⊢ Y ∈ C
  H  ⊢ g ∈ C(X,Y)
```

We have generated rule objects for all the rules described in Section 2, as well as rules for dealing with products. Logical rules and rules dealing with extensional equality and substitution are already provided by Nuprl.

For each inference rule, we have also proved formally that it is correct with respect to our formalization of basic category theory. Although this is not strictly necessary if one is mainly interested in automating proofs, it validates the consistency of the implemented inference system relative to the consistency of Nuprl.

4 Proof Automation

The implementation of the proof calculus described above enables us to create formal proofs for many theorems of basic category theory. But even the simplest such theorems lead to proofs with hundreds or even thousands of inference steps, as illustrated in [18]. Since most of these statements are considered mathematically trivial, it should be possible to find their proofs completely automatically.

We have developed strategies for automated proof search in basic category theory that attempt to capture the general patterns of reasoning that we have observed in hand-constructed proofs. In this section we discuss the key components of these strategies and some of the issues that had to be reckoned with.

Automated Rule Application. Most of the inference rules are simple refinements that describe how to decompose a proof obligation into simpler components. Given a specific proof goal, there are only few rules that can be applied at all. Thus to a large extent, proof search consists of determining applicable rules and their parameters from the context, applying the rule, and then continuing the search on all the subgoals.

To make this possible, all the basic inference rules had to be converted into simple tactics that automatically determine their parameters. Generating names for new variables in the subgoals, as in the case of the extensionality rules (17)–(20), is straightforward. All other parameters occur as types in one of the subgoals of a rule and are determined through an extended type inference algorithm.

An important issue is loop control. Since the synthesis rules for functors and natural transformations are the inverse of the corresponding analysis rules, we have to avoid applying analysis rules if they create a subgoal that has already been decomposed by a synthesis rule. Synthesis rules decrease the depth of functor types in a proof goal. It is therefore sufficient to keep track of proof goals to which a synthesis rule had been applied and block the application of analysis rules that would generate one of these as a subgoal.

Performance Issues. One of the disadvantages of refinement style reasoning is that proof trees may contain identical proof goals in different branches. This is especially true after the application of synthesis and extensionality rules, which must be used quite often in complex proofs. The first subgoal of rule (9) eventually reappears in the proof of the second, since F^1A occurs within the type of that goal and both subgoals reappear in the proofs of the third and fourth subgoals. In a bottom-up proof, one would prove these goals only once and reuse them whenever they are needed to complete the proof of another goal, while a standard refinement proof forces us to prove the same goal over and over again.

Obviously we could optimize the corresponding rules for top-down reasoning and drop the redundant subgoals. To retain faithfulness of the implemented inference system, however, we decided to leave the rules unchanged. Instead, we have wrapped the corresponding tactic with a controlled application of the cut rule: we assert the first two subgoals of rule (9) before applying the rule. As a result they appear as hypotheses of all subgoals and have to be proved only once.

Although this method is a simple trick, it leads to a significant reduction in the size of automatically generated proofs. A complete proof of the isomorphism between $\mathsf{Fun[C \times D, E]}$ and $\mathsf{Fun[C, Fun[D, E]]}$ without cuts consists of almost 30,000 inference steps. Using the wrapper reduces this number to 3,000.

Equality Reasoning. Equality reasoning is a key component in formal category-theoretic proofs. Ten of the inference rules deal with equalities and can be used to replace a term by one that is semantically equal.

Since equalities can be used both ways, they can easily lead to infinite loops in an automated search for a proof. Our automated reasoning strategy therefore has to assign a direction to each of the equalities and attempt to rewrite terms into some normal form. Furthermore, it has to keep track of the types involved in these equalities, which are sometimes crucial for finding a proper match and, as in the case of rule (15), for determining the right-hand side of an equality from the left-hand side. The inference rules described in Section 2, including those dealing with associativity and identity, lead to the following typed rewrites.

Rewrite			**Type**	**Rule**
$<g,k>\circ<f,h>$	$\mapsto$	$<g\circ f, k\circ h>$	$\mathsf{C{\times}D}(<A_1,X_1>,<A_3,X_3>)$	(01)
$1_{<A,X>}$	$\mapsto$	$<1_A, 1_X>$	$\mathsf{C{\times}D}(<A,X>,<A,X>)$	(02)
$1_Y\circ f$	$\mapsto$	f	$\mathsf{C}(X,Y)$	(2a)
$f\circ 1_X$	$\mapsto$	f	$\mathsf{C}(X,Y)$	(2b)
$h\circ(g\circ f)$	$\mapsto$	$(h\circ g)\circ f$	$\mathsf{C}(X,T)$	(2c)
$F^2(g\circ f)$	$\mapsto$	$F^2 g\circ F^2 f$	$\mathsf{D}(F^1X, F^1Z)$	(07)
$F^2 1_X$	$\mapsto$	1_{F^1X}	$\mathsf{D}(F^1X, F^1X)$	(08)
$(\psi\circ\varphi)A$	$\mapsto$	$\psi A\circ\varphi A$	$\mathsf{D}(F^1X, H^1X)$	(11)
$1_F X$	$\mapsto$	1_{F^1X}	$\mathsf{D}(F^1X, F^1X)$	(12)
$\varphi Y\circ F^2 g$	$\mapsto$	$G^2 g\circ\varphi X$	$\mathsf{D}(F^1X, G^1Y)$	(15)

Each rewrite is executed by applying a substitution, which is validated by applying the corresponding equality rule mentioned in the table above.

The above rewrite system is incomplete, as it cannot prove the equality of some terms that can be shown equal with the inference rules. We have used the superposition-based Knuth-Bendix completion procedure [17] to generate the following additional typed rewrites.

Rewrite			**Type**	**Rules**
$F^2<1_A, 1_X>$	$\mapsto$	$1_{F^1<A,X>}$	$\mathsf{E}(F^1<A,X>, F^1<A,X>)$	(02),(08)
$F^2<g,k>\circ F^2<f,h>$	$\mapsto$	$F^2<g\circ f, k\circ h>$	$\mathsf{E}(F^1<A,X>, F^1<C,X>)$	(01),(07)
$(\varphi YA)\circ(F^2 gA)$	$\mapsto$	$(G^2 gA)\circ(\varphi XA)$	$\mathsf{E}(F^1X^1A, G^1Y^1A)$	(15),(11)
$(\varphi Y\circ\psi Y)\circ F^2 g$	$\mapsto$	$(G^2 g\circ\varphi X)\circ\psi X$	$\mathsf{E}(F^1X, G^1Y)$	(11),(15)
$H^2(\varphi Y)\circ H^2(F^2 g)$	$\mapsto$	$H^2(G^2 g)\circ H^2(\varphi X)$	$\mathsf{E}(H^1F^1X, H^1G^1Y)$	(2c),(15)
$(h\circ\varphi Y)\circ F^2 g$	$\mapsto$	$(h\circ G^2 g)\circ\varphi X$	$\mathsf{D}(F^1X, Z)$	(07),(15)
$((h\circ G^2 g)\circ\varphi X)\circ\psi X$	$\mapsto$	$((h\circ\varphi Y)\circ\psi Y)\circ F^2 g$	$\mathsf{E}(F^1X, Z)$	(2c),(11),(15)
$(h\circ H^2(\varphi Y))\circ H^2(F^2 g)$	$\mapsto$	$(h\circ H^2(G^2 g))\circ H^2(\varphi X)$	$\mathsf{E}(H^1F^1X, Z)$	(07),(07),(15)

First-Order Reasoning. One important aspect of our approach is demonstrating that reasoning in basic category theory is essentially first-order although some of its concepts are not. This means that functors and natural transformations can only be treated as abstract objects whose properties can only be described in terms of their first-order components.

For example, proving two categories C and D isomorphic (formally denoted by $\mathtt{C} \mathrel{\hat{=}} \mathtt{D}$) requires showing the existence of two functors θ : $\mathsf{Fun}\,\mathtt{[}\mathsf{C},\mathsf{D}\mathtt{]}$ and η : $\mathsf{Fun}\,\mathtt{[}\mathsf{D},\mathsf{C}\mathtt{]}$ that are inverses of each other. In the formal proof, we cannot simply introduce θ as closed object, because this would be a pair of λ-terms mapping C-objects onto D-objects and C-arrows onto D-arrows. Instead we have to specify its object and arrow components $\theta^1 A$ and $\theta^2 f$ for A an object of C and f an arrow of C through first-order equations. If these components are again functors or natural transformations, we have to specify subcomponents until we have reached a first-order level. In our proof of the isomorphism between $\mathsf{Fun}\,\mathtt{[}\mathsf{C}\times\mathsf{D},\mathsf{E}\mathtt{]}$ and $\mathsf{Fun}\,\mathtt{[}\mathsf{C},\mathsf{Fun}\,\mathtt{[}\mathsf{D},\mathsf{E}\mathtt{]]}$, we need four equations to specify θ:

$$\begin{array}{lll}
\theta^1\mathtt{G}^1\mathtt{X}^1\mathtt{X1} \equiv \mathtt{G}^1\mathtt{<X,\ X1>} & \qquad & \theta^1\mathtt{G}^2\mathtt{f\ X} \equiv \mathtt{G}^2\mathtt{<f,\ 1X>} \\
\theta^1\mathtt{G}^1\mathtt{X}^2\mathtt{h} \equiv \mathtt{G}^2\mathtt{<1X,\ h>} & \qquad & \theta^2\,\varphi\ \mathtt{X\ X1} \equiv \varphi\ \mathtt{<X,\ X1>}
\end{array}$$

Mathematically speaking, these four equations are sufficient for the proof, since any functor satisfying these equations can be used to complete the proof. However, the embedding of basic category theory into Nuprl's formal logic requires the existence of a functor satisfying these equations to be verified (this requirement could, of course, be turned off by providing a special rule). Constructing the functor from the equations is straightforward if it is uniquely specified by them. Since this part of the proof is higher-order and has nothing to do with basic category theory, it is generated automatically in the background.

Guessing Witnesses for Existential Quantifiers. The mechanisms described so far are sufficient to verify properties of given functors and natural transformations. But many proofs in basic category theory require proving the existence of functors or transformations with certain properties. For a trained mathematician this is a trivial task if there are only few "obvious" choices. Since the purpose of proof automation is automating what is considered obvious, we have developed a heuristic that attempts to determine specifications for functors or natural transformations that are most likely to make a proof succeed.

The most obvious approach is to start developing a proof where the functor or natural transformation has been replaced by a free variable and proceed until the decomposition cannot continue anymore. At this point we have generated typing subgoals for all first-order components of the functor. For the functor θ in our isomorphism proof we get (up to α-equality) four different typing conditions

$$\begin{array}{ll}
\theta^1\mathtt{G}^1\mathtt{X}^1\mathtt{X1} \in \mathtt{E} & \theta^1\mathtt{G}^2\mathtt{f\ X} \in \mathtt{E(}\theta^1\mathtt{G}^1\mathtt{A}^1\mathtt{X,\ }\theta^1\mathtt{G}^1\mathtt{B}^1\mathtt{X)} \\
\theta^1\mathtt{G}^1\mathtt{X}^2\mathtt{h} \in \mathtt{E(}\theta^1\mathtt{G}^1\mathtt{X}^1\mathtt{X1,\ }\theta^1\mathtt{G}^1\mathtt{X}^1\mathtt{Y)} & \theta^2\,\varphi\ \mathtt{X\ X1} \in \mathtt{E(}\theta^1\mathtt{F}^1\mathtt{X}^1\mathtt{X1,\ }\theta^1\mathtt{G}^1\mathtt{X}^1\mathtt{X\ X1)}
\end{array}$$

The heuristic then tries to determine the simplest term that is built from the component's parameters (whose types are known) and satisfies the given typing requirements. For this purpose it tries to identify parameters that are declared to be functors or natural transformations and to find a match between some part of their range type and the typing requirement for the component. Once the match has been found, the remaining parameters will be used to determine the arguments needed by the functor or natural transformation.

To solve the first of the above typing conditions, the heuristic finds the declarations $G : \mathsf{Fun}\,[\mathsf{C} \times \mathsf{D}, \mathsf{E}]$, $X : \mathsf{C}$, and $X_1 : \mathsf{D}$. The simplest term that has type E and is built from these parameters is the term $G^1\texttt{<}X\,\texttt{,}X_1\texttt{>}$.

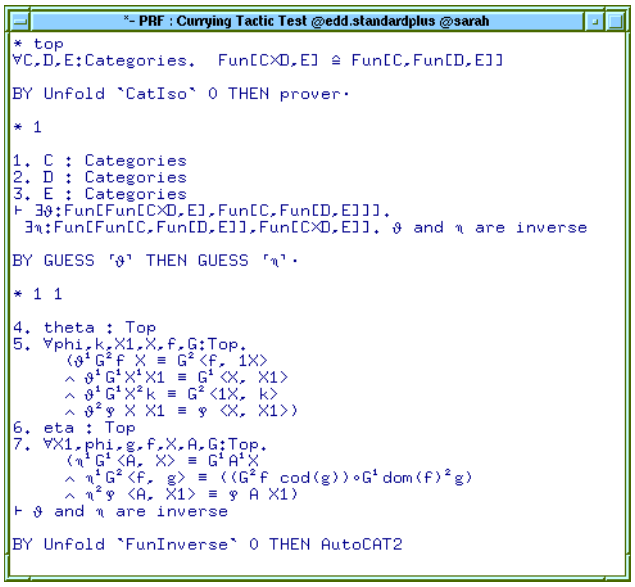
Determining the arguments of a functor or natural transformation is not always straightforward. In some cases like the above, the remaining parameters are of the right type and can be used as arguments. In other cases we have an object where an arrow is needed or vice versa. The most obvious choice is turning an object into an identity arrow and an arrow into its domain or codomain, depending on the typing requirements.

To solve the second of the above conditions, the heuristic has to use the declarations $G : \mathsf{Fun}\,[\mathsf{C} \times \mathsf{D}, \mathsf{E}]$, $X : \mathsf{C}$, $h : \mathsf{D}(X_1, Y)$, and $X_1, Y : \mathsf{D}$. To create a term of type $\mathsf{E}(\theta^1 G^1 X^1 X_1, \theta^1 G^1 X^1 Y)$, one has to use G^2 and arrows from $\mathsf{C}(X, X)$ and $\mathsf{D}(X_1, Y)$. For the latter, we can pick h, while the only arrow in $\mathsf{C}(X, X)$ that can be built from the object X is the identity 1_X.

In some cases, none of the above choices satisfy the typing conditions, but a composition of natural transformation and functor as in rule (15) would do so. In this case, the heuristic will use the functor and its arguments twice in different ways. This choice is less obvious, but still considered standard.

5 An Application

To demonstrate the feasibility of our approach, we have generated a completely formal proof that the functor categories $\mathsf{Fun}\,[\mathsf{C} \times \mathsf{D}, \mathsf{E}]$ and $\mathsf{Fun}\,[\mathsf{C}, \mathsf{Fun}\,[\mathsf{D}, \mathsf{E}]]$



```
* top
∀C,D,E:Categories.  Fun[C×D,E] ≙ Fun[C,Fun[D,E]]

BY Unfold `CatIso` 0 THEN prover·

* 1

1. C : Categories
2. D : Categories
3. E : Categories
⊢ ∃ϑ:Fun[Fun[C×D,E],Fun[C,Fun[D,E]]].
  ∃η:Fun[Fun[C,Fun[D,E]],Fun[C×D,E]]. ϑ and η are inverse

BY GUESS ⌜ϑ⌝ THEN GUESS ⌜η⌝·

* 1 1

4. theta : Top
5. ∀phi,k,X1,X,f,G:Top.
      (ϑ^1 G^2 f X ≡ G^2 <f, 1X>
      ∧ ϑ^1 G^1 X^1 X1 ≡ G^1 <X, X1>
      ∧ ϑ^1 G^1 X^2 k ≡ G^2 <1X, k>
      ∧ ϑ^2 φ X X1 ≡ φ <X, X1>)
6. eta : Top
7. ∀X1,phi,g,f,X,A,G:Top.
      (η^1 G^1 <A, X> ≡ G^1 A^1 X
      ∧ η^1 G^2 <f, g> ≡ ((G^2 f cod(g))∘G^1 dom(f)^2 g)
      ∧ η^2 φ <A, X1> ≡ φ A X1)
⊢ ϑ and η are inverse

BY Unfold `FunInverse` 0 THEN AutoCAT2
```

are naturally isomorphic. The structure of the proof is similar to the hand-constructed proof described in [18], which required several hours to complete and more than 10 pages to write down. Using our strategies, the creation of the proof was fully automated and took only a few seconds.

The screenshot above shows that the proof consists of only six proof steps. First we unfold the definition of isomorphisms and decompose the proof goal. We then ask the tactic to guess values for the functors θ and η. Finally, we unfold the definition of inverse functors and use the automated proof search procedure to validate that θ and η are indeed functors of the appropriate types and that they are inverse to each other.

This top-level version of the proof reveals the key idea that was necessary to solve the problem, but hides the tedious details involved in validating the solution. Users interested in proof details can inspect the complete proof tree that Nuprl will display on demand. However, one should be aware that the complete proof is huge. It takes 1046 and 875 basic inferences to prove that θ and η are indeed functors of the appropriate types and another 1141 inferences to prove that they are inverse to each other.

It should be noted that all six steps are straightforward when it comes to dealing with isomorphism problems. One could combine them into a single tactic `IsoCAT`, which would then give us the following proof.

```
* ∀C,D,E:Categories.  Fun[C×D,E] ≙ Fun[C,Fun[D,E]]
  BY IsoCAT
```

However, little insight is gained from such a proof, except that it has in fact been completed automatically.

Proving the naturality of the isomorphism is more demanding, since we have to show θ and η to be elements of $\mathsf{Fun}[\mathsf{Cat}^{\mathsf{op}} \times \mathsf{Cat}^{\mathsf{op}} \times \mathsf{Cat}, \mathsf{Cat}](U, V)$ for suitable functors U, V, that are inverse for every choice of categories C, D, and E. Guessing specifications for U, V, θ, and η automatically is now less trivial. Currently, our automated strategy (extended for dealing with categories of categories) can only validate U, V, θ, and η after they have been specified by hand.

6 Conclusion

We have presented a Gentzen-style deductive system for elementary category theory involving a mixture of typing and equational judgments. We have implemented this logic in Nuprl along with relevant proof tactics that go a long way toward full automation of elementary proofs. We have demonstrated the effectiveness of this approach by automatically deriving proofs of several nontrivial results in the theory, one example of which is presented in detail above. The system works very well on the examples we have tried.

We have found that careful planning in the order of application of tactics makes the proof search mostly deterministic. However, the proofs that are generated tend to be quite large because of the overwhelming amount of detail. Many of the necessary steps, especially those that involve basic typing judgments, are quite tedious and do not lend much insight from a human perspective. For this reason, they are typically omitted in the literature. Such arguments are nevertheless essential for automation, because they drive the application of tactics.

There are a number of technical insights that we have observed in the course of this work.

- Most of the ideas that we have applied in this work are in fact fairly standard and not too sophisticated. This shows that our calculus is well designed and integrates well with existing theorem proving technology.
- Formal proofs, even of quite elementary facts, have thousands of inferences. As mentioned, many of these steps are quite tedious and do not lend much insight. This indicates that the theory is a good candidate for automation.
- Almost all proof steps can be automated. Forward steps such as decomposition using the analysis rules and directed rewriting for equations tend to be quite successful. Since the proof system is normalizing and confluent (we did not show this), the time is mostly spent building the proof. Apart from guessing witnesses, there is virtually no backtracking involved and the bulk of the development is completely deterministic, being driven by typing considerations.
- Lookahead improves the performance of our strategy. Since inference rules may generate redundant subgoals, lemma generation can allow proof reuse.
- Display forms are crucial for comprehensibility. It is often very difficult to keep track of typing judgments currently in force. Judicious choice of the display form can make a great difference in human readability.

For the future, we plan to gain more experience by attempting to automate more of the basic theory. We need more experience with the different types of arguments that arise in category theory so that we will be better able to design those parts of the mechanism involved with the guessing of witnesses. Preliminary investigations show that automating the application of the Yoneda lemma will be key to many of the more advanced proofs.

Since our proof strategies can be viewed as encodings of proof plans for category theory, our approach may benefit from using generic proof planning techniques [8] to make these proof plans explicit.

Finally, we would like to mention an intriguing theoretical open problem. The proof of the result that we have described, namely that $\mathsf{Fun}\,[\mathsf{C} \times \mathsf{D}, \mathsf{E}]$ and $\mathsf{Fun}\,[\mathsf{C}, \mathsf{Fun}\,[\mathsf{D}, \mathsf{E}]\,]$ are naturally isomorphic, breaks down into two parts. The first part argues that the functor categories $\mathsf{Fun}\,[\mathsf{C} \times \mathsf{D}, \mathsf{E}]$ and $\mathsf{Fun}\,[\mathsf{C}, \mathsf{Fun}\,[\mathsf{D}, \mathsf{E}]\,]$ are isomorphic, and the second part argues that the isomorphism is natural. As Mac Lane describes it [20, p. 2], *naturality*, applied to a parameterized construction, says that the construction is carried out "in the same way" for all instantiations of the parameters. Of course, there is a formal definition of the concept of naturality in category theory itself, and it involves reparameterizing the result in terms of functors in place of objects, natural transformations in place of arrows. But any constructions in the formal proof π of the first part of the theorem, just the isomorphism of the two parameterized functor categories, would work "in the same way" for all instantiations of the parameters, by virtue of the fact that the formal proof π is similarly parameterized. This leads us to ask: Under what conditions can one extract a proof of naturality *automatically* from π? That is, under what conditions can a proof in our formal system be automatically retooled to additionally establish the naturality of the constructions involved? Extracting naturality in this way would be analogous to the extraction of programs from proofs according to the Curry–Howard isomorphism.

References

1. S. Allen et. al. Innovations in computational type theory using Nuprl. *Journal of Applied Logic*, 2006. (to appear).
2. S. Allen, R. Constable, R. Eaton, C. Kreitz, L. Lorigo The Nuprl open logical environment. *CADE-17*, LNCS 1831, pages 170–176. Springer, 2000.
3. G. Bancerek. Concrete categories. *J. formalized mathematics*, 13, 2001.
4. G. Bancerek. Miscellaneous facts about functors. *J. form. math.*, 13, 2001.
5. G. Bancerek. Categorial background for duality theory. *J. form. math.*, 13, 2001.
6. M. Barr, C. Wells. *Category Theory for Computing Science.* Prentice Hall, 1990.
7. R. Bird. *A Calculus of Functions for Program Derivation*, Research Topics in Functional Programming, pages 287–307, Addison-Wesley, 1990.
8. A. Bundy *The Use of Explicit Plans to Guide Inductive Proofs*. *CADE-9*, pages 111–120. Springer, 1988.
9. R. Constable et. al. *Implementing Mathematics with the Nuprl proof development system.* Prentice Hall, 1986.
10. M. J. Cáccamo, G. Winskel. A higher-order calculus for categories. Technical Report RS-01-27, BRICS, University of Aarhus, 2001.
11. P. Eklund et. al. A graphical approach to monad compositions. *Electronic Notes in Theoretical Computer Science*, 40, 2002.
12. S. Eilenberg, S. MacLane. General theory of natural equivalences. *Trans. Amer. Math. Soc.*, 58:231–244, 1945.
13. J. Glimming. Logic and automation for algebra of programming. Master thesis, University of Oxford, 2001.
14. J. Goguen. A categorical manifesto. *Mathematical Structures in Computer Science*, 1(1):49–67, 1991.
15. J. Harrison. Formalized mathematics. *Technical report of Turku Centre for Computer Science*, 36, 1996.
16. G. Huet, A. Saïbi. Constructive category theory. *Joint CLICS-TYPES Workshop on Categories and Type Theory*, 1995. MIT Press.
17. D. Knuth, P. Bendix. Simple word problems in universal algebra. *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.
18. D. Kozen. Toward the automation of category theory. Technical Report 2004-1964, Computer Science Department, Cornell University, 2004.
19. C. Kreitz. *The Nuprl Proof Development System, V5: Reference Manual and User's Guide.* Computer Science Department, Cornell University, 2002.
20. S. MacLane. *Categories for the Working Mathematician.* Springer, 1971.
21. E. Moggi. Notions of computation and monads. *Inf. and Comp.*, 93, 1991.
22. Greg O'Keefe. Towards a readable formalisation of category theory. *Electronic Notes in Theoretical Computer Science* 91:212–228. Elsevier, 2004.
23. L. Paulson. *Isabelle: A Generic Theorem Prover*, LNCS 828. Springer, 1994.

24. D. Rydeheard, R. Burstall. *Computational Category Theory.* Prentice Hall, 1988.
25. G. M. Reed, A. W. Roscoe, R. F. Wachter. *Topology and Category Theory in Computer Science.* Oxford University Press, 1991.
26. A. Saïbi. Constructive category theory, coq.inria.fr/contribs/category.tar.gz, 1995.
27. C. Simpson. Category theory in ZFC, coq.inria.fr/contribs/CatsInZFC.tar.gz, 2004.
28. M. Takeyama. *Universal Structure and a Categorical Framework for Type Theory.* PhD thesis, University of Edinburgh, 1995.
29. A. Trybulec. Some isomorphisms between functor categories. *J. formalized mathematics*, 4, 1992.
30. P. Wadler. Monads for functional programming. *Advanced Functional Programming*, LNCS 925, pp. 24–52, Springer, 1995.