

# Nuprl as Logical Framework for Automating Proofs in Category Theory

Christoph Kreitz

Institut für Informatik, Universität Potsdam, 14482 Potsdam, Germany

**Abstract.** We describe the construction of a semi-automated proof system for elementary category theory using the Nuprl proof development system as logical framework. We have used Nuprl’s display mechanism to implement the basic vocabulary and Nuprl’s rule compiler to implement a first-order proof calculus for reasoning about categories, functors and natural transformations. To automate proofs we have formalized both standard techniques from automated theorem proving and reasoning patterns that are specific to category theory and used Nuprl’s tactic mechanism for the actual implementation. We illustrate our approach by automating proofs of natural isomorphisms between categories.

## 1 Introduction

Category theory [EM45] is a common framework for expressing abstract properties of mathematical structures that occur in many areas of mathematics and computer science. Abstract notions such as objects, morphisms, composition, identities, products, functors, transformations, duality, and isomorphisms are common to areas like set theory, logic, algebra, topology, semantics of programming languages, or formal software specification and development. The beauty of category theory is that it allows one to be completely precise about such concepts and that many algebraic constructions become exceedingly elegant at this level of abstraction. Diagrams can be used to illustrate essential insights and often make it unnecessary to provide further details of a proof, as these may be obtained entirely by standard patterns of reasoning.

However, since category theory is considerably more abstract than many other branches of mathematics, it becomes almost impossible to verify the details of such a proof. Readers frequently have to accept “obvious” assertions on faith, as complete proofs based on precise definitions often involve an enormous number of low-level details that must be checked. Furthermore, the high level of abstraction forces one to work in an atmosphere in which much of the intuition has been stripped away. As a result, the verification often becomes a matter of pure symbol manipulation, an area in which humans easily make mistakes.

On the other hand, proofs that rely on standard patterns of reasoning and symbol manipulation lend themselves well to automation. Providing such an automation serves several purposes. It enables users to generate completely formal proofs without having to go through all the details themselves, thus providing

assurance that the statement is in fact true. It allows users to inspect details of a proof and get a better grasp of the standard patterns of reasoning in elementary category theory. It also shows that the proofs that many authors do not bother to provide actually may contain a tremendous amount of hidden detail and possibly even preconditions that the author might have taken for granted or overlooked entirely. Finally, it demonstrates that a proof is indeed trivial from an intellectual point of view, because it could be found automatically by a machine.

To provide a foundation for automating basic category theory reasoning Kozen [Koz04] presents a first-order axiomatization of elementary category theory and illustrates its use by giving a formal proof that the functor categories  $\text{Fun}[C \times D, E]$  and  $\text{Fun}[C, \text{Fun}[D, E]]$  are naturally isomorphic. Although the proof of this theorem omits many low-level details such as equality reasoning and simple first-order arguments, it is extremely long and required many hours of careful work to complete. To make sure that every detail of a proof can be validated it is necessary to implement the proof calculus and to develop proof strategies that support the automated construction of proofs by capturing the general patterns of reasoning used in hand-constructed proofs.

As platform for the implementation of Kozen's calculus we have selected the Nuprl system [CA<sup>+</sup>86]. Nuprl is a proof and programming environment for the interactive development of formalized mathematical knowledge as well as for the synthesis, verification, and optimization of software. Nuprl's current architecture [AC<sup>+</sup>00, Kre02, AB<sup>+</sup>05] is the product of many evolutions aimed at providing a theorem proving environment as rich and robust as its type theory. The resulting implementation composes a set of communicated processes, centered around a common knowledge base, called the *library*. The library contains definition objects, theorems, inference rules, and meta-level code (e.g. tactics), and serves as a transaction broker for the other processes. Those processes include user interfaces (*editors*), inference engines (*refiners*) and mechanisms for extracting programs from proofs, rewrite engines (*evaluators*), and *translators*. Translators between the formal knowledge stored in the library and, for instance, programming languages like Java or Ocaml [Kre04, KHH98] allow the formal reasoning tools to supplement real-world software from various domains and thus provide a *logical programming environment* for the respective languages.

While Nuprl was originally developed as theorem prover for Computational Type Theory [Con08], the current architecture has no predefined logic but uses formal library objects to define the syntax and inference rules of a logic. Thus the Nuprl system has become a *logical framework* that can accommodate arbitrary logics whose inference rules can be expressed in a sequent style. Although almost all of the actual development is still based on the Nuprl type theory, users may now embed entirely new theories as independent proof calculi into the system's library and use the framework to automate reasoning in these theories.

To make use of this potential of the Nuprl system, which had not been explored before, we proceeded as follows. To embed the vocabulary of elementary category theory we added abstract terms for each concept of the theory to the system's library as well as display forms for presenting these terms in a familiar

syntax. For all inference rules of Kozen’s calculus we added rule objects to the library and used Nuprl’s rule compiler to convert these into reasoning tactics that execute these rules in the proof environment. To automate reasoning we encoded standard theorem proving techniques as Nuprl proof tactics, developed additional tactics to capture the reasoning patterns that are specific to category theory, and added these tactics as code objects to the library.

In [KKR06] we have demonstrated that this approach can in fact automate Kozen’s proof of the isomorphism between  $\text{Fun}[C \times D, E]$  and  $\text{Fun}[C, \text{Fun}[D, E]]$ . With the additional techniques and calculi described in this paper, we are now able to construct fully automated proofs for a variety of isomorphisms between categories as well as proofs of the naturality of all all these isomorphisms.

The structure of this paper follows the development outlined above. In Section 2 we briefly review Kozen’s first-order axiomatization of elementary category theory. We then describe the implementation of this calculus within the Nuprl logical framework in Section 3. Strategies that encode standard techniques from automated theorem proving will be presented in Section 4. Strategies that automate reasoning specific to category theory and their application to proofs of natural isomorphisms will be discussed in Section 5. We conclude by discussing related approaches, insights that we have observed in the course of this work, and new research issues that result from these observations.

## 2 An Axiomatization of Elementary Category Theory

We assume the reader to be familiar with the basic definitions and notation of category theory [BW90, McL71]. We begin our review of Kozen’s calculus [Koz04] with a few notational conventions.

- Symbols in sans serif, such as  $C$ , always denote categories. The category  $\text{Cat}$  is the category of (small) categories and functors.
- If  $C$  is a category, the symbol  $\mathbf{C}$  denotes both the category  $C$  and the set of *objects* of  $C$ .
- $A : C$  indicates that  $A$  is an object of the category  $C$ . Composition is denoted by the symbol  $\circ$  and the identity on object  $A : C$  is denoted  $\mathbf{1}_A$ .
- $h : C(A, B)$  indicates that  $h$  is an *arrow* of the category  $C$  with *domain*  $A$  and *codomain*  $B$ .
- $\text{Fun}[C, D]$  denotes the functor category whose objects are the *functors* from  $C$  to  $D$  and whose arrows are the *natural transformations* on such functors. Thus  $F : \text{Fun}[C, D]$  indicates that  $F$  is a functor from  $C$  to  $D$  and  $\varphi : \text{Fun}[C, D](F, G)$  indicates that  $\varphi$  is a natural transformation with domain  $F$  and codomain  $G$ , where  $F, G : \text{Fun}[C, D]$ .
- $F^1$  and  $F^2$  denote the object and arrow components, respectively, of a functor  $F$ . Thus if  $F : \text{Fun}[C, D]$ ,  $A, B : C$ , and  $h : C(A, B)$ , then  $F^1 A, F^1 B : D$  and  $F^2 h : D(F^1 A, F^1 B)$ .
- Function application binds tighter than the operators  $^1$  and  $^2$ . Thus the expression  $F^1 A^2$  should be parsed  $(F^1 A)^2$ .

- $\mathbf{C}^{\text{op}}$  denotes the opposite category of  $\mathbf{C}$ .
- $\mathbf{C} \times \mathbf{D}$  denotes the product of the categories  $\mathbf{C}$  and  $\mathbf{D}$ . Its objects are pairs  $(A, X) : \mathbf{C} \times \mathbf{D}$ , where  $A : \mathbf{C}$  and  $X : \mathbf{D}$ , and its arrows consist of pairs  $(f, h) : (\mathbf{C} \times \mathbf{D})((A, X), (B, Y))$ , where  $f : \mathbf{C}(A, B)$  and  $h : \mathbf{D}(X, Y)$ . Composition and identities are defined component-wise; that is,

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

$$\mathbf{1}_{(A, X)} \stackrel{\text{def}}{=} (\mathbf{1}_A, \mathbf{1}_X). \quad (2)$$

Inference rules are based on sequents  $\Gamma \vdash \alpha$ , where  $\Gamma$  is a type environment (a set of type judgments on atomic symbols) and  $\alpha$  is either a type judgment or an equation. The rules cover the basic properties of categories, functors and natural transformations. They are divided into symmetric sets of rules for analysis (elimination) and synthesis (introduction). There are also rules for equational reasoning. To support first-order reasoning about higher-order concepts like functors, the rules deal with their first-order components.

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

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

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

$$\frac{\Gamma \vdash A, B, C, D : \mathbf{C}, \quad \Gamma \vdash f : \mathbf{C}(A, B), \quad \Gamma \vdash g : \mathbf{C}(B, C), \quad \Gamma \vdash h : \mathbf{C}(C, D)}{\Gamma \vdash (h \circ g) \circ f = h \circ (g \circ f)} \quad (5)$$

$$\frac{\Gamma \vdash A, B : \mathbf{C}, \quad \Gamma \vdash f : \mathbf{C}(A, B)}{\Gamma \vdash f \circ \mathbf{1}_A = f} \quad \frac{\Gamma \vdash A, B : \mathbf{C}, \quad \Gamma \vdash f : \mathbf{C}(A, B)}{\Gamma \vdash \mathbf{1}_B \circ f = f} \quad (6)$$

**Functors.** A functor  $F$  from  $\mathbf{C}$  to  $\mathbf{D}$  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*

$$\frac{\Gamma \vdash F : \text{Fun}[\mathbf{C}, \mathbf{D}], \quad \Gamma \vdash A : \mathbf{C}}{\Gamma \vdash F^1 A : \mathbf{D}} \quad (7)$$

$$\frac{\Gamma \vdash F : \text{Fun}[\mathbf{C}, \mathbf{D}], \quad \Gamma \vdash A, B : \mathbf{C}, \quad \Gamma \vdash f : \mathbf{C}(A, B)}{\Gamma \vdash F^2 f : \mathbf{D}(F^1 A, F^1 B)} \quad (8)$$

$$\frac{\Gamma \vdash F : \text{Fun}[\mathbf{C}, \mathbf{D}], \quad \Gamma \vdash A, B, C : \mathbf{C}, \quad \Gamma \vdash f : \mathbf{C}(A, B), \quad \Gamma \vdash g : \mathbf{C}(B, C)}{\Gamma \vdash F^2(g \circ f) = F^2 g \circ F^2 f} \quad (9)$$

$$\frac{\Gamma \vdash F : \text{Fun}[\mathbf{C}, \mathbf{D}], \quad \Gamma \vdash A : \mathbf{C}}{\Gamma \vdash F^2 1_A = 1_{F^1 A}} \quad (10)$$

*Synthesis*

$$\frac{\begin{array}{c} \Gamma, A : \mathbf{C} \vdash F^1 A : \mathbf{D} \\ \Gamma, A, B : \mathbf{C}, g : \mathbf{C}(A, B) \vdash F^2 g : \mathbf{D}(F^1 A, F^1 B) \\ \Gamma, A, B, C : \mathbf{C}, f : \mathbf{C}(A, B), g : \mathbf{C}(B, C) \vdash F^2(g \circ f) = F^2 g \circ F^2 f \\ \Gamma, A : \mathbf{C} \vdash F^2 1_A = 1_{F^1 A} \end{array}}{\Gamma \vdash F : \text{Fun}[\mathbf{C}, \mathbf{D}]} \quad (11)$$

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

$$\begin{array}{ccc} F^1 A & \xrightarrow{F^2 g} & F^1 B \\ \varphi A \downarrow & & \downarrow \varphi B \\ G^1 A & \xrightarrow{G^2 g} & G^1 B \end{array} \quad (12)$$

Composition and identities are defined by

$$(\varphi \circ \psi) A \stackrel{\text{def}}{=} \varphi A \circ \psi A \quad (13)$$

$$1_{F A} \stackrel{\text{def}}{=} 1_{F^1 A}. \quad (14)$$

The property (12), along with the typing of  $\varphi$ , are captured in the following rules.

*Analysis*

$$\frac{\Gamma \vdash \varphi : \text{Fun}[\mathbf{C}, \mathbf{D}](F, G)}{\Gamma \vdash F, G : \text{Fun}[\mathbf{C}, \mathbf{D}]} \quad (15)$$

$$\frac{\Gamma \vdash \varphi : \text{Fun}[\mathbf{C}, \mathbf{D}](F, G), \quad \Gamma \vdash A : \mathbf{C}}{\Gamma \vdash \varphi A : \mathbf{D}(F^1 A, G^1 A)} \quad (16)$$

$$\frac{\Gamma \vdash \varphi : \text{Fun}[\mathbf{C}, \mathbf{D}](F, G), \quad \Gamma \vdash A, B : \mathbf{C}, \quad \Gamma \vdash g : \mathbf{C}(A, B)}{\Gamma \vdash \varphi B \circ F^2 g = G^2 g \circ \varphi A} \quad (17)$$

*Synthesis*

$$\frac{\begin{array}{c} \Gamma \vdash F, G : \text{Fun}[\mathbf{C}, \mathbf{D}] \\ \Gamma, A : \mathbf{C} \vdash \varphi A : \mathbf{D}(F^1 A, G^1 A) \\ \Gamma, A, B : \mathbf{C}, g : \mathbf{C}(A, B) \vdash \varphi B \circ F^2 g = G^2 g \circ \varphi A \end{array}}{\Gamma \vdash \varphi : \text{Fun}[\mathbf{C}, \mathbf{D}](F, G)} \quad (18)$$

**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 (5) and two-sided identity rules (6) for arrows, the equations (1) and (2) for products, and the equations (13) and (14) for natural transformations. There are also extensionality rules for objects of functional type:

$$\frac{\Gamma \vdash F, G : \text{Fun}[\mathbf{C}, \mathbf{D}], \quad \Gamma, A : \mathbf{C} \vdash F^1 A = G^1 A}{\Gamma \vdash F^1 = G^1} \quad (19)$$

$$\frac{\Gamma \vdash F, G : \text{Fun}[\mathbf{C}, \mathbf{D}], \quad \Gamma, A, B : \mathbf{C}, \quad g : \mathbf{C}(A, B) \vdash F^2 g = G^2 g}{\Gamma \vdash F^2 = G^2} \quad (20)$$

$$\frac{\Gamma \vdash F, G : \text{Fun}[\mathbf{C}, \mathbf{D}], \quad \Gamma \vdash F^1 = G^1, \quad \Gamma \vdash F^2 = G^2}{\Gamma \vdash F = G} \quad (21)$$

$$\frac{\Gamma \vdash F, G : \text{Fun}[\mathbf{C}, \mathbf{D}], \quad \Gamma \vdash \varphi, \psi : \text{Fun}[\mathbf{C}, \mathbf{D}](F, G), \quad \Gamma, A : \mathbf{C} \vdash \varphi A = \psi A}{\Gamma \vdash \varphi = \psi} \quad (22)$$

Equations on types and substitution of equals for equals in type expressions are also permitted. Any such equation  $\alpha = \beta$  takes the form of a rule

$$\frac{\Gamma \vdash A : \alpha}{\Gamma \vdash A : \beta}. \quad (23)$$

For the application example, the following type equations are postulated

$$\text{Cat}(\mathbf{C}, \mathbf{D}) = \text{Fun}[\mathbf{C}, \mathbf{D}] \quad (24)$$

$$\mathbf{C}^{\text{op}} = \mathbf{C} \quad (25)$$

$$\mathbf{C}^{\text{op}}(A, B) = \mathbf{C}(B, A). \quad (26)$$

**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 Implementing the Proof Calculus in Nuprl

Kozen's axiomatization is sufficient for the development of completely formal proofs for all theorems in elementary category theory. Kozen [Koz04] illustrates this fact by providing a rigorous formal proof that the functor categories  $\text{Fun}[\mathbf{C} \times \mathbf{D}, \mathbf{E}]$  and  $\text{Fun}[\mathbf{C}, \text{Fun}[\mathbf{D}, \mathbf{E}]]$  are naturally isomorphic. Although the proof is fairly straightforward and omits many details for the sake of readability it takes 13 pages on paper. As proofs of that size are difficult to construct and carry the potential for errors it is necessary to implement the proof system and to automate reasoning steps that mathematicians would consider obvious. In this section we will show how the Nuprl logical framework can be used to rapidly construct an implementation of Kozen's calculus.

### 3.1 Embedding the Vocabulary

We made use of Nuprl’s definition mechanism to implement the vocabulary of elementary category theory. *Abstraction objects* can be used to add new abstract terms to the formal library whose meaning may either be defined through expressions of the formal language defined so far, or be left unspecified if only the signature shall be fixed. The abstract term for the set of objects of a category  $C$ , for instance, is implemented by adding an abstraction object named `Obj` to the library, which in the library listing appears as follows.

```
A Obj      Obj{.C} == !primitive
```

This object introduces a new abstract term `Obj` with one subterm, denoted by the variable  $C$ , and defines it to be primitive.<sup>1</sup>

*Display forms* can then be used to make the visual appearance of abstract terms conform to the notation used on paper without changing their internal structure. According to the conventions in section 2, for instance, the set of objects of a category  $C$  is denoted by  $C$ . To introduce this notation, we add a display object named `Obj_df` to the library.

```
D Obj_df    <C> ≡ Obj{.<C>}
```

This object makes sure that the abstract term `Obj{.C}` will be displayed as  $C$ . The angle brackets around  $C$  indicate that  $C$  is a parameter of the display form.

Display forms are important for interaction between the system and human users as well as for a readable presentation of the implemented theory on paper. The reasoning system itself deals only with abstract terms and can therefore easily distinguish between a category and the set of its objects although both have the same representation on the screen.

We have created abstraction and display objects for each concept of elementary category theory. Although it is possible to represent these concepts in terms of Nuprl’s Type Theory and validate the implemented inference rules on this basis we have chosen to use Nuprl only as logical framework for building an independent system for reasoning about category theory and have declared all fundamental abstract terms to be primitive.

Figure 1 lists the display objects that implement the vocabulary of elementary category theory. For the sake of readability we use math-font instead of angle brackets to denote parameters. Note that a composition  $g \circ f$  of two arrows depends on the category  $C$  to which  $f$  and  $g$  belong but  $C$  is never mentioned explicitly in compositions. Accordingly,  $C$  has to occur in the abstract term but should not be shown when a composition is being displayed. For this reason, the parameter  $C$  does not occur on the left hand side of the display form for compositions and identities and the display of the category will be suppressed as soon as the complete term has been entered into the system.

<sup>1</sup> The formal declaration of `Obj` also contains an empty list of parameters between curly braces and an empty list of variable bindings for the subterm in front of the dot before  $C$ . Parameters and variable bindings are features of the Nuprl logical framework that are necessary for representing more expressive theories but are not needed for implementing elementary category theory.

Obj	$C$	$\equiv$	Obj{ }( $.C$ )
Mor	$C(A, B)$	$\equiv$	Mor{ }( $.C; .A; .B$ )
Comp	$(g \circ f)$	$\equiv$	Comp{ }( $.C; .g; .f$ )
Id	$1A$	$\equiv$	Id{ }( $.C; .A$ )
fun1	$F^1A$	$\equiv$	fun1{ }( $.F; .A$ )
fun2	$F^2g$	$\equiv$	fun2{ }( $.F; .g$ )
CatFun	$\text{Fun}[C, D]$	$\equiv$	CatFun{ }( $.C; .D$ )
CatProd	$C \times D$	$\equiv$	CatProd{ }( $.C; .D$ )
CatOp	$C\text{-op}$	$\equiv$	CatOp{ }( $.C$ )
CatCat	$\text{Cat}$	$\equiv$	CatCat{ }()

**Fig. 1.** Display objects implementing the syntax of elementary category theory

Currently, Nuprl's display is restricted to a single 8-bit font. This limits the use of symbols, subscripts and superscripts to fixed characters. Identities, usually written as  $1_A$  or  $1_{(A, X)}$ , have to be presented as  $1A$  and  $1\langle A, X \rangle$ .<sup>2</sup> Apart from these restrictions, all the basic category-theoretic vocabulary will be displayed in the same way as described in Section 2.

Besides the vocabulary of elementary category theory Kozen [Koz04] uses a notion of isomorphism of categories and naturality of isomorphisms. These concepts can be defined in terms of the existing notions (see Section 5.2 for an explanation) and are implemented by the following formal definitions.

*F and G are inverse*  
 $\equiv \forall A, B: C. \forall f: C(A, B). G^1 F^1 A = A \in C \wedge G^2 F^2 f = f \in C(A, B)$   
 $\wedge \forall X, Y: D. \forall h: D(X, Y). F^1 G^1 X = X \in D \wedge F^2 G^2 h = h \in D(X, Y)$

*C  $\hat{=}$  D*  
 $\equiv \exists \theta: \text{Fun}[C, D]. \exists \eta: \text{Fun}[D, C]. \theta \text{ and } \eta \text{ are inverse}$

*C  $\hat{=}$  D via  $\theta$  and  $\eta$*   
 $\equiv \theta \in \text{Fun}[C, D] \wedge \eta \in \text{Fun}[D, C] \wedge \theta \text{ and } \eta \text{ are inverse}$

*C and D are naturally isomorphic*  
 $\equiv \exists \text{CAT}. \exists U, V: \text{Fun}[\text{CAT}, \text{Cat}]$   
 $\exists \theta: \text{Fun}[\text{CAT}, \text{Cat}](U, V). \exists \eta: \text{Fun}[\text{CAT}, \text{Cat}](V, U).$   
 $\forall c: \text{CAT}. C \hat{=} D \text{ via } \theta_c \text{ and } \eta_c$

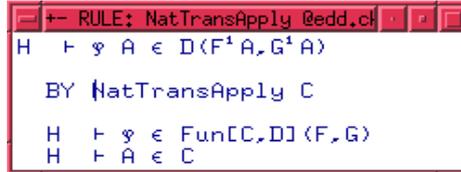
### 3.2 Implementation of Inference Rules

Like the proof calculus presented in the previous section, Nuprl's inference mechanism is based on sequents. Nuprl's reasoning style, however, is goal-oriented, which means that inference rules operate top-down, refining a goal sequent into a set of subgoal sequents. Inference rules therefore have to be rephrased in a top-down fashion before they can be added to the system.

For the actual implementation of the proof calculus we made use of Nuprl's rule mechanism. *Rule objects* can be used to add schematic inference rules to the formal library. These consist of formal terms that describe a goal sequent and

<sup>2</sup> In Nuprl pairs use angle brackets  $\langle A, X \rangle$  instead of parentheses.

the corresponding subgoal sequents and may contain pattern variables that can be matched against the components of the actual goal sequent in a proof. Since the representation of the rules in the system is identical to the paper version it is easy to check the faithfulness of the implementation. Rule (16), for instance, is represented by a rule object `NatTransApply` with the following contents.



```

+- RULE: NatTransApply @edd,cl
H ⊢ φ A ∈ D(F¹ A, G¹ A)

BY NatTransApply C

H ⊢ φ ∈ Fun[C, D]⟨F, G⟩
H ⊢ A ∈ C

```

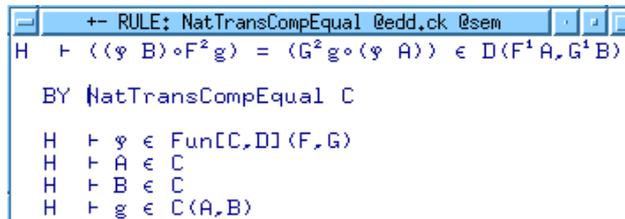
The rule states that in order to prove a goal sequent  $\Gamma \vdash \varphi A : D(F^1 A, G^1 A)$  one has to prove  $\Gamma \vdash \varphi : \text{Fun}[C, D](F, G)$  and  $\Gamma \vdash A : C$  for some category  $C$ , which is exactly the same as rule (16). Due to the top-down style of the rule,  $C$  must be provided as parameter, since it occurs in the subgoal sequents but not in the main goal.

To create the actual inference rule from its representation as term one applies the *rule compiler* of the Nuprl logical framework to the rule object. This generates a proof tactic that matches the first line of a rule object against the actual goal sequent of a proof and creates the subgoal sequents by instantiating the lines below the name of the rule accordingly. The proof tactic for the above rule, for instance, is generated by the following simple ML declaration

```
let NatTransApply C = Refine 'NatTransApply' [term_arg C].
```

To apply this tactic, one has to provide a term  $C$  as argument, which is then inserted into the two subgoals created by the rule. Tactics may also expect tokens as arguments, which will then be used as names for variables that occur in the subgoals but not in the main goal. The tactic for the synthesis rule (11), for instance, requires five such names (for  $A, B, C, f$ , and  $g$ ) to be provided.

Since Nuprl supports typed equalities and types often provide useful information for guiding proofs, we added types to all the inference rules that deal with equalities. For example, rule (17) is represented as follows:



```

+- RULE: NatTransCompEqual @edd,ck @sem
H ⊢ ((φ B) ∘ F² g) = (G² g ∘ (φ A)) ∈ D(F¹ A, G¹ B)

BY NatTransCompEqual C

H ⊢ φ ∈ Fun[C, D]⟨F, G⟩
H ⊢ A ∈ C
H ⊢ B ∈ C
H ⊢ g ∈ C⟨A, B⟩

```

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.

## 4 Automating First-Order and Equational Reasoning

The implementation of the proof calculus described in Section 3 enables us to create formal proofs for many theorems of basic category theory. But even the most simple of these theorems already lead to proofs with hundreds or even thousands of inference steps, as illustrated in [Koz04]. Since most of these statements are considered mathematically trivial, it should be possible to completely automate their proofs.

We have developed a small collection of strategies for automated proof search in basic category theory. Some of these strategies are based on generic techniques from automated theorem proving. Others are intended to capture the general patterns of category theory specific reasoning that we have observed in hand-constructed proofs. We will discuss the former in this section and elaborate on the latter in Section 5.

Most of the inference rules of our proof calculus are simple refinement rules 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. Occasionally we will have to prove equalities, which may involve the application of extensionality rules as well as standard equality reasoning.

### 4.1 Automating Search

To support proof automation, all basic inference rules first had to be converted into simple tactics that automatically determine the parameters of these rules.

Generating names for new variables in the subgoals, as in the case of the extensionality rules (19)–(22), is straightforward. In principle it is sufficient to use a procedure that generate arbitrary new names but for the sake of readability we had the procedure generate mnemonic names that fit the textual description of the rules.

To determine the terms that have to be provided as parameters for certain inference rules one can take advantage of the fact that these parameters are explicitly mentioned in the subgoals of the rule, which puts certain type constraints on possible values. In the rule `NatTransApply`, for instance, `C` is the category to which the object `A` belongs and also the domain of the functors `F` and `G`. Therefore all term parameters of inference rules can be determined through an *extended type inference* algorithm.

To identify applicable rules it is sufficient to analyze the terms and types in the conclusion of the goal sequent. A conclusion of the form  $\varphi A \in D(X, Y)$ , for instance, suggests the application of the rule `NatTransApply` or, less likely, of the rule `NatTransFormation` (rule (18)) if `D` turns out to be a functor category. In most cases only one rule can be meaningfully applied to a proof goal with a type judgment and this rule can be identified with the help of extended type inference.

An important issue is *loop control*. Since the synthesis rules for functors and natural transformations are the inverse of the corresponding analysis rules, an analysis rule could create a subgoal that has already been decomposed by a synthesis rule before and thus create a looping argument. To prevent such loops we made each proof branch keep track of proof goals to which a synthesis rule had been applied. Analysis rules that would generate one of these goals as a subgoal will thus be blocked from being applied.

## 4.2 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 equality rules can be used both ways, they are a very powerful tool in the hands of a skilled user, but a potential cause for loops in an automated search for a proof. A simple proof search method as described above is therefore insufficient for automating proofs involving equalities.

We have decided to base our proof search mechanism on rewriting. For the purpose of *finding* a proof for a given equality we assign a direction to each of the equalities and attempt to rewrite terms into some normal form. Furthermore, the search procedure 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 (17), 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
$\langle g, k \rangle \circ \langle f, h \rangle \mapsto \langle g \circ f, k \circ h \rangle$	$C \times D(\langle A_1, X_1 \rangle, \langle A_3, X_3 \rangle)$	(01)
$1_{\langle A, X \rangle} \mapsto \langle 1_A, 1_X \rangle$	$C \times D(\langle A, X \rangle, \langle A, X \rangle)$	(02)
$1_B \circ f \mapsto f$	$C(A, B)$	(06a)
$f \circ 1_A \mapsto f$	$C(A, B)$	(06b)
$h \circ (g \circ f) \mapsto (h \circ g) \circ f$	$C(A, B_2)$	(05)
$F^2(g \circ f) \mapsto F^2 g \circ F^2 f$	$D(F^1 A, F^1 B_1)$	(09)
$F^2 1_A \mapsto 1_{F^1 A}$	$D(F^1 A, F^1 A)$	(10)
$(\psi \circ \varphi) A \mapsto \psi A \circ \varphi A$	$D(F^1 A, H^1 A)$	(13)
$1_{F A} \mapsto 1_{F^1 A}$	$D(F^1 A, F^1 A)$	(14)
$\varphi B \circ F^2 g \mapsto G^2 g \circ \varphi A$	$D(F^1 A, G^1 B)$	(17)

Each rewrite is executed by applying a substitution, which is validated by applying the corresponding equality rule mentioned in the table above. The equations (24)–(26) deal solely with types and are treated separately.

The above rewrite system is incomplete, as it cannot prove the equality of terms like  $F^2(1_A, 1_X)$  and  $1_{F^1(A, X)}$  that can be shown equal with the inference rules. To convert the equational theory contained in our calculus into an equivalent set of rewrite rules guaranteeing normalization and confluence, we have applied the superposition-based Knuth-Bendix completion procedure [EB70]. As a result, the following typed rewrites were added to the system.

Rewrite		Type	Rules
$F^2\langle 1_A, 1_X \rangle$	$\mapsto 1_{F^1\langle A, X \rangle}$	$E(F^1\langle A, X \rangle, F^1\langle A, X \rangle)$	(02), (10)
$F^2\langle g, k \rangle \circ F^2\langle f, h \rangle$	$\mapsto F^2\langle g \circ f, k \circ h \rangle$	$E(F^1\langle A, X \rangle, F^1\langle C, X \rangle)$	(01), (09)
$(\varphi Y A) \circ (F^2 g A)$	$\mapsto (G^2 g A) \circ (\varphi X A)$	$E(F^1 X^1 A, G^1 Y^1 A)$	(17), (13)
$(\varphi Y \circ \psi Y) \circ F^2 g$	$\mapsto (G^2 g \circ \varphi X) \circ \psi X$	$E(F^1 X, G^1 Y)$	(13), (17)
$H^2(\varphi Y) \circ H^2(F^2 g)$	$\mapsto H^2(G^2 g) \circ H^2(\varphi X)$	$E(H^1 F^1 X, H^1 G^1 Y)$	(05), (17)
$(h \circ \varphi Y) \circ F^2 g$	$\mapsto (h \circ G^2 g) \circ \varphi X$	$D(F^1 X, Z)$	(09), (17)
$((h \circ G^2 g) \circ \varphi X) \circ \psi X$	$\mapsto ((h \circ \varphi Y) \circ \psi Y) \circ F^2 g$	$E(F^1 X, Z)$	(05), (13), (17)
$(h \circ H^2(\varphi Y)) \circ H^2(F^2 g)$	$\mapsto (h \circ H^2(G^2 g)) \circ H^2(\varphi X)$	$E(H^1 F^1 X, Z)$	(09), (09), (17)

Once a set of rewrites for a given equality has been found it is converted into a series of refinement steps by applying the equality rules associated with each rewrite in the appropriate direction. As a result, the generated proof tree contains a trace of the chain of equalities used, which can then be inspected by a human user interested in understanding the details of a proof.

### 4.3 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.

```

H ⊢ F ∈ Fun[C,D]
BY FunFormation A B B1 f g
  H, A:C ⊢ F1A ∈ D
  H, A:C, B:C, f:C(A,B) ⊢ F2f ∈ D(F1A, F1B)
  H, A:C, B:C, B1:C, f:C(A,B), g:C(B, B1) ⊢ F2(g ∘ f) = F2g ∘ F2f ∈ D(F1A, F1B1)
  H, A:C ⊢ F21A = 1F1A ∈ D(F1A, F1A)
    
```

The rule **FunFormation** (rule (11)), for instance, generates a subgoal of the form  $F^1A$ , which will eventually reappear in the proof of the second, since  $F^1A$  occurs within the type of that goal. Furthermore, the first two subgoals will also 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 would require us to prove the same goal over and over again.

Obviously we could optimize the corresponding rules for top-down reasoning and simply drop the redundant subgoals. But this would mean deviating from the original proof calculus. If one intends to retain faithfulness these rules must remain unchanged. Instead, we have wrapped the corresponding tactic with a controlled application of the cut rule: we simply assert a generalization of the first two subgoals of rule (11) before applying the rule. As a result they appear in the hypothesis list of the all subgoals and have to be proved only once.

Although this method is a fairly simple trick, it leads to an astonishing reduction in the size of automatically generated proofs. A complete proof of the isomorphism between  $\text{Fun}[C \times D, E]$  and  $\text{Fun}[C, \text{Fun}[D, E]]$  (see Section 5) without cuts consists of almost 30,000 inference steps. After introducing the wrapper the size of the proof was reduced to only 3,000 steps.

#### 4.4 First-Order Reasoning about Higher Order Objects

Although elementary category theory deals with higher-order objects such as functors and natural transformations, Kozen's axiomatization [Koz04] has been formulated entirely as first-order calculus. This means that the properties of functors and natural transformations have to be described in terms of their first-order components (rules (7) – (11), (15) – (18)).

Keeping the reasoning level first order becomes more difficult when reasoning about isomorphisms between categories. Two categories  $\mathbf{C}$  and  $\mathbf{D}$  are isomorphic, denoted by  $\mathbf{C} \cong \mathbf{D}$ , if there are two functors  $\theta : \text{Fun}[\mathbf{C}, \mathbf{D}]$  and  $\eta : \text{Fun}[\mathbf{D}, \mathbf{C}]$  that are inverses of each other. A computerized proof of this fact would require us to provide  $\theta$  and  $\eta$ , which involves higher-order reasoning.

To avoid this issue, the proof in [Koz04] specifies the object and arrow components  $\theta^1 A$  and  $\theta^2 f$  for  $A$  an object of  $\mathbf{C}$  and  $f$  an arrow of  $\mathbf{C}$  through first-order equations. If these components are again functors or natural transformations, one has to specify subcomponents until the first-order level has been reached. In the proof of the isomorphism between  $\text{Fun}[\mathbf{C} \times \mathbf{D}, \mathbf{E}]$  and  $\text{Fun}[\mathbf{C}, \text{Fun}[\mathbf{D}, \mathbf{E}]]$ , the following four equations are needed to specify  $\theta$ :

$$\begin{aligned} \theta^1 F^1 A^1 X &\equiv F^1 \langle A, X \rangle \\ \theta^1 F^1 A^2 k &\equiv F^2 \langle 1A, k \rangle \\ \theta^1 F^2 f X &\equiv F^2 \langle f, 1X \rangle \\ \theta^2 \varphi X X1 &\equiv \varphi \langle X, X1 \rangle \end{aligned}$$

Mathematically, these four equations are sufficient for the proof, since any functor satisfying these equations can be used to complete the proof. In a computerized formal proof, however, we also have to prove the existence of a functor satisfying these equations. Constructing the functor from the equations is straightforward if it is uniquely specified by them. It is only necessary to assemble the respective object and arrow (sub-)components into a single closed functor object. Since assembling the functor from components has nothing to do with the main proof, this step is performed automatically in the background as soon as the components have been completely specified.

## 5 Automating Reasoning Specific to Category Theory

The mechanisms described in the previous section are sufficient to verify properties of given functors and natural transformations. A proof of the isomorphism between  $\text{Fun}[\mathbf{C} \times \mathbf{D}, \mathbf{E}]$  and  $\text{Fun}[\mathbf{C}, \text{Fun}[\mathbf{D}, \mathbf{E}]]$  can be completely automated once the specifications of the inverse functors  $\theta$  and  $\eta$  have been provided. One only has to unfold the definition of functors being inverse to each other and then all the remaining steps are straightforward for the automated proof search procedure `AutoCAT2` and take only a few seconds to complete.

```

*  $\forall C, D, E: \text{Categories. } \text{Fun}[C \times D, E] \hat{=} \text{Fun}[C, \text{Fun}[D, E]]$ 
BY ....
1.-3.  $C, D, E: \text{Cat}$ 
4.  $\theta^1 F^1 A^1 X \equiv F^1 \langle A, X \rangle \wedge \theta^1 F^1 A^2 k \equiv F^2 \langle 1A, k \rangle$ 
    $\wedge \theta^1 F^2 f X \equiv F^2 \langle f, 1X \rangle \wedge \theta^2 \varphi X X1 \equiv \varphi \langle X, X1 \rangle$ 
5.  $\eta^1 F^1 \langle A, X \rangle \equiv F^1 A^1 X \wedge \eta^1 F^2 \langle f, g \rangle \equiv ((F^2 f \text{ cod}(g)) \circ F^1 \text{dom}(f))^2 g$ 
    $\wedge \eta^2 \varphi \langle A, X1 \rangle \equiv \varphi A X1$ 
 $\vdash \theta$  and  $\eta$  are inverse
BY AutoCAT2

```

But finding the specifications of the functors cannot be accomplished with standard reasoning techniques, since matching and unification are of little help here. On the other hand, for a trained mathematician this is a trivial task as there are only a few “obvious” choices. If a functor exists at all the types of its first-order components usually contain all the information that is needed to make an educated guess. In practice, this simple heuristic hardly ever fails, particularly if the proof is considered trivial from an intellectual point of view.

Since the proof steps that are considered intellectually trivial should be automated, we have developed heuristics that attempt to determine the most obvious specifications for functors or natural transformations of a given type. We will illustrate both by example of the isomorphism between the categories  $\text{Fun}[C \times D, E]$  and  $\text{Fun}[C, \text{Fun}[D, E]]$  and the naturality of this isomorphism and then discuss a few details of their formalization as implemented proof strategy.

### 5.1 Finding Witnesses for Isomorphisms between Categories

To prove the existence of a functor  $F$  between two categories  $C$  and  $D$  our heuristic first generates typing subgoals for all first-order components of the functor. For this purpose it applies the refinement rules of our proof calculus to the goal  $\Gamma \vdash F \in \text{Fun}[C, D]$ , where  $\Gamma$  is the current context of the proof and  $F$  is a new variable, and proceeds with refining typing judgments until they cannot be decomposed anymore. Equalities will be ignored as they do not provide information that is immediately useful.

To prove the existence of a functor  $\theta$  between the categories  $\text{Fun}[C \times D, E]$  and  $\text{Fun}[C, \text{Fun}[D, E]]$ , for instance, the application of refinement rules yields the (incomplete) proof shown in figure 2. The four open subgoals in this proof, labelled 1.1.1, 1.1.2, 1.2.1, and 2.1.1, describe the typing conditions for all the first-order components of  $\theta$ .

Next, the heuristic tries to determine a term that satisfies the given type judgment in the corresponding type environment. Because this term is intended to be a “trivial” solution, it should use be built solely from parameters explicitly mentioned in the first-order component of the functor and constructs that mathematicians would consider obvious choices like functor application, identities, domains, ranges, pairs etc. Obviously, the heuristic has to rely on type inference to construct a term that fits these requirements.

To solve subgoal 1.1.1., for instance, the heuristic has to construct an object of the category  $E$  from the components  $F : \text{Fun}[C \times D, E]$ ,  $A : C$ , and  $X : D$ .

$$\begin{array}{ll}
\text{top} & \vdash \theta \in \text{Fun}[\text{Fun}[\text{C}\times\text{D},\text{E}], \text{Fun}[\text{C},\text{Fun}[\text{D},\text{E}]]] & (11) \\
1. & F:\text{Fun}[\text{C}\times\text{D},\text{E}] \quad \vdash \theta^1 F \in \text{Fun}[\text{C},\text{Fun}[\text{D},\text{E}]] & (11) \\
1.1. & F:\text{Fun}[\text{C}\times\text{D},\text{E}], A:\text{C} \vdash \theta^1 F^1 A \in \text{Fun}[\text{D},\text{E}] & (11) \\
1.1.1. & F:\text{Fun}[\text{C}\times\text{D},\text{E}], A:\text{C}, X:\text{D} & \\
& \quad \vdash \theta^1 F^1 A^1 X \in \text{E} & \\
1.1.2. & F:\text{Fun}[\text{C}\times\text{D},\text{E}], A:\text{C}, X,Y:\text{D}, k:\text{D}(X,Y) & \\
& \quad \vdash \theta^1 F^1 A^2 k \in \text{E}(\theta^1 F^1 A^1 X, \theta^1 F^1 A^1 Y) & \\
1.2. & F:\text{Fun}[\text{C}\times\text{D},\text{E}], A,B:\text{C}, f:\text{C}(A,B), X:\text{D} & \\
& \quad \vdash \theta^1 F^2 f \in \text{Fun}[\text{D},\text{E}] (\theta^1 F^1 A, \theta^1 F^1 B) & (18) \\
1.2.1. & F:\text{Fun}[\text{C}\times\text{D},\text{E}], A,B:\text{C}, f:\text{C}(A,B), X:\text{D} & \\
& \quad \vdash \theta^1 F^2 f X \in \text{E}(\theta^1 F^1 A^1 X, \theta^1 F^1 B^1 X) & \\
2. & F,G:\text{Fun}[\text{C}\times\text{D},\text{E}], \varphi:\text{Fun}[\text{C}\times\text{D},\text{E}](F,G) & \\
& \quad \vdash \theta^2 \varphi \in \text{Fun}[\text{C},\text{Fun}[\text{D},\text{E}]] (\theta^1 F, \theta^1 G) & (18) \\
2.1. & F,G:\text{Fun}[\text{C}\times\text{D},\text{E}], \varphi:\text{Fun}[\text{C}\times\text{D},\text{E}](F,G), X:\text{C} & \\
& \quad \vdash \theta^2 \varphi X \in \text{Fun}[\text{D},\text{E}] (\theta^1 F^1 X, \theta^1 G^1 X) & (18) \\
2.1.1. & F,G:\text{Fun}[\text{C}\times\text{D},\text{E}], \varphi:\text{Fun}[\text{C}\times\text{D},\text{E}](F,G), X:\text{C}, X1:\text{D} & \\
& \quad \vdash \theta^2 \varphi X X1 \in \text{E}(\theta^1 F^1 X^1 X1, \theta^1 G^1 X^1 X1) & 
\end{array}$$

**Fig. 2.** Decomposition of the typing  $\theta \in \text{Fun}[\text{Fun}[\text{C}\times\text{D},\text{E}], \text{Fun}[\text{C},\text{Fun}[\text{D},\text{E}]]]$ . The numbers on the right indicate the inference rules that were used.

Among the declared parameters, there is no object of the category  $\text{E}$ , so the heuristic looks for parameters whose types contain the goal type. It finds the functor  $F$  with range  $\text{E}$ , which reduces the task of constructing an object of  $\text{E}$  to constructing an object  $z : \text{C}\times\text{D}$  and applying  $F^1$  to it. Since objects in  $\text{C}\times\text{D}$  are pairs  $(A, X)$  where  $A : \text{C}$  and  $X : \text{D}$ , the task is now finding an object in  $\text{C}$  and one in  $\text{D}$ . There are obvious choices for these two objects in the type environment, which means that all the components of the term have been identified. As a result, the heuristic returns the specification  $\theta^1 F^1 A^1 X = F^1(A, X)$ .

Determining the arguments of a functor or natural transformation is not always as straightforward. In the above case, the parameters in the type environment could be taken directly as components of the term because their types fit the requirements on these components. In other cases, the type environment may provide only an object where an arrow is needed or vice versa. In these situations 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 subgoal 1.1.2. we have to build an arrow in  $\text{E}(\theta^1 F^1 A^1 X, \theta^1 F^1 A^1 Y)$  from the components  $F : \text{Fun}[\text{C}\times\text{D},\text{E}]$ ,  $A : \text{C}$ ,  $X, Y : \text{D}$ , and  $k : \text{D}(X, Y)$ . Since subgoal 1.1.1. has already been solved, the equality  $\theta^1 F^1 A^1 X = F^1(A, X)$  can be used to simplify the goal type to  $\text{E}(F^1(A, X), F^1(A, Y))$ . To build an arrow of that type from the given parameters, the heuristic has to apply  $F^2$  to an arrow  $h \in \text{C}\times\text{D}((A, X), (A, Y))$ , i.e. to a pair of arrows  $(f, g)$  where  $f : \text{C}(A, A)$  and  $g : \text{D}(X, Y)$ . For the latter, we can pick  $k$  but there is no immediate match for  $f : \text{C}(A, A)$ . Since  $\theta^1 F^1 A^2 k$ , the component of  $\theta$  that shall be specified in this step, explicitly mentions  $A$ , the only choice for an arrow in  $\text{C}(A, A)$  is the identity  $1_A$ . As a result, the heuristic returns the specification  $\theta^1 F^1 A^2 k = F^2(1_A, k)$ .

Subgoal 1.2.1. can be solved in the same manner, which leads to the specification  $\theta^1 F^2 f X = F^2(f, 1_X)$ . The solution of subgoal 2.1.1. proceeds as the one for subgoal 1.1.1. and yields  $\theta^2 \varphi X X_1 = \varphi(X, X_1)$ .

In some cases, parameters and identities alone are not sufficient to satisfy a typing conditions, but a simple composition of natural transformation and functor in the style of the equality rule (17) would do so. In this case, the heuristic has to use the functor and its arguments twice in different ways. Although this solution is less obvious it is still considered a standard pattern of reasoning.

We have developed a *calculus for witness construction* that formalizes the heuristic described above. Rules decompose a goal sequent similar to our proof rules in section 3.2 and come with a mechanism that composes sets of specification equations for subgoal sequents into specification equations for the main goal.

There are a few rules that construct specification equations from scratch. If an element  $z$  of type  $\Delta$  has been declared then it can be used as witness to satisfy the type judgment  $x \in \Delta$ , i.e. we construct the specification equation  $x = z$ . Furthermore, if the declaration contains a type  $\Delta[V_1, ..V_n]$  with free type variables  $V_i$  and the type judgment contains an instantiated version  $\Delta[T_1, ..T_n]$ , then the equations  $V_1 = T_1, .., V_n = T_n$  will be constructed as well. In our calculus we write this rule as follows

$$\Gamma, z:\Delta[V_1, ..V_n] \vdash x \in \Delta[T_1, ..T_n] \quad \text{specs } \{x = z, V_1 = T_1, .., V_n = T_n\},$$

where the notation **specs**  $EQ$  indicates that the set of specification equations  $EQ$  will be constructed if the rule can be applied successfully.

There is also a rule that constructs identities to satisfy a type judgment  $f \in C(A, A)$  if the type environment contains a declaration of an object  $A$  of  $C$  and a rule that constructs domains or codomains to satisfy a type judgment  $x \in C$  if the type environment contains a declaration of an arrow  $f \in C(A, B)$ .

Other rules decompose category constructors like functors or products that occur in the type environment or in the type judgment. For instance, in order to use a functor  $F:Fun[C,D]$  when constructing a term  $x \in \Delta$  one has to construct an object  $z$  of  $C$  and show how to use an object  $y$  of  $D$  in the construction of  $x$ . If both goals succeed and yield specification equations  $EQ_1$  and  $EQ_2$  then the specification equation for the main goal is the union of  $EQ_1$  and  $EQ_2$  where  $y$  is being replaced by  $F^1 z$ .

$$\begin{array}{l} \Gamma, F:Fun[C,D] \vdash x \in \Delta \\ \Gamma \vdash z \in C \\ \Gamma, y : D \vdash x \in \Delta \end{array} \quad \begin{array}{l} \text{specs } EQ_1 \cup EQ_2[F^1 z/y] \\ \text{specs } EQ_1 \\ \text{specs } EQ_2 \end{array}$$

For each constructor there are two rules for decomposing objects and arrows in a type judgment and two rules for decomposing objects and arrows in the type environment. There are also equality reduction rules that simplify a type judgment or a part of the environment by applying a known equality. Figure 3 shows the fragment of our calculus that is necessary for dealing with sequents containing functors and products.

## 1. Basic rules:

– $\Gamma, z: \Delta[V_1, \dots, V_n] \vdash x \in \Delta[T_1, \dots, T_n]$	<b>specs</b> $\{x = z, V_1 = T_1, \dots, V_n := T_n\}$
– $\Gamma, A: \mathcal{C} \vdash f \in \mathcal{C}(A, A)$	<b>specs</b> $\{f = 1_A\}$
– $\Gamma, f: \mathcal{C}(A, B) \vdash x \in \mathcal{C}$	<b>specs</b> $\{x = \text{dom}(f)\}$
– $\Gamma, f: \mathcal{C}(A, B) \vdash x \in \mathcal{C}$	<b>specs</b> $\{x := \text{cod}(f)\}$
– $\Gamma, f: \mathcal{C}(A, B) \vdash h \in \mathcal{C}(A, X)$	<b>specs</b> $\{h = g \circ f\} \cup EQ$
– $\Gamma, f: \mathcal{C}(A, B) \vdash h \in \mathcal{C}(A, X)$	<b>specs</b> $EQ$
– $\Gamma, g: \mathcal{C}(B, X) \vdash h \in \mathcal{C}(A, X)$	<b>specs</b> $\{h = g \circ f\} \cup EQ$
– $\Gamma, g: \mathcal{C}(B, X) \vdash f \in \mathcal{C}(A, B)$	<b>specs</b> $EQ$
– $\Gamma, \text{exp} = t \vdash x \in \Delta$	<b>specs</b> $EQ$
– $\Gamma, \text{exp} = t \vdash x \in \Delta[t/\text{exp}]$	<b>specs</b> $EQ$
– $\Gamma, \text{exp} = t, y: \Gamma' \vdash x \in \Delta$	<b>specs</b> $EQ$
– $\Gamma, \text{exp} = t, y: \Gamma'[t/\text{exp}] \vdash x \in \Delta$	<b>specs</b> $EQ$

## 2. Functors:

– $\Gamma \vdash F \in \text{Fun}[\mathcal{C}, \mathcal{D}]$	<b>specs</b> $EQ_1 \cup EQ_2$
– $\Gamma, c: \mathcal{C} \vdash F^1 c: \mathcal{D}$	<b>specs</b> $EQ_1$
– $\Gamma, EQ_1, f: \mathcal{C}(A, B) \vdash F^2 f \in \mathcal{D}(F^1 A, F^1 B)$	<b>specs</b> $EQ_2$
– $\Gamma \vdash \varphi \in \text{Fun}[\mathcal{C}, \mathcal{D}](F, G)$	<b>specs</b> $EQ$
– $\Gamma, A: \mathcal{C} \vdash \varphi A \in \mathcal{D}(F^1 A, G^1 A)$	<b>specs</b> $EQ$
– $\Gamma, F: \text{Fun}[\mathcal{C}, \mathcal{D}] \vdash x \in \Delta$	<b>specs</b> $EQ_1 \cup EQ_2[F^1 z/y]$
– $\Gamma \vdash z \in \mathcal{C}$	<b>specs</b> $EQ_1$
– $\Gamma, y: \mathcal{D} \vdash x \in \Delta$	<b>specs</b> $EQ_2$
– $\Gamma, F: \text{Fun}[\mathcal{C}, \mathcal{D}] \vdash x \in \Delta$	<b>specs</b> $EQ_1 \cup EQ_2[F^2 f/h]$
– $\Gamma \vdash f \in \mathcal{C}(T_1, T_2)$	<b>specs</b> $EQ_1$
– $\Gamma, A, B: \mathcal{C}, h: \mathcal{D}(F^1 A, F^1 B) \vdash x \in \Delta$	<b>specs</b> $EQ_2 \cup \{A = T_1, B = T_2\}$
– $\Gamma, \varphi: \text{Fun}[\mathcal{C}, \mathcal{D}](F, G) \vdash x \in \Delta$	<b>specs</b> $EQ_1 \cup EQ_2[\varphi A/h]$
– $\Gamma \vdash A \in \mathcal{C}$	<b>specs</b> $EQ_1$
– $\Gamma, h: \mathcal{D}(F^1 A, G^1 A) \vdash x \in \Delta$	<b>specs</b> $EQ_2$

## 3. Products:

– $\Gamma \vdash z \in \mathcal{C} \times \mathcal{D}$	<b>specs</b> $EQ_1 \cup EQ_2 \cup \{z = \langle c, d \rangle\}$
– $\Gamma \vdash c \in \mathcal{C}$	<b>specs</b> $EQ_1$
– $\Gamma \vdash d \in \mathcal{D}$	<b>specs</b> $EQ_2$
– $\Gamma \vdash f \in \mathcal{C} \times \mathcal{D}(\langle A, X \rangle, \langle B, Y \rangle)$	<b>specs</b> $EQ_1 \cup EQ_2 \cup \{f = \langle g, h \rangle\}$
– $\Gamma \vdash g \in \mathcal{C}(A, B)$	<b>specs</b> $EQ_1$
– $\Gamma \vdash h \in \mathcal{D}(X, Y)$	<b>specs</b> $EQ_2$
– $\Gamma, z: \mathcal{C} \times \mathcal{D} \vdash x \in \Delta$	<b>specs</b> $EQ \cup \{z = \langle c, d \rangle\}$
– $\Gamma, c: \mathcal{C}, d: \mathcal{D} \vdash x \in \Delta$	<b>specs</b> $EQ$
– $\Gamma, f: \mathcal{C} \times \mathcal{D}(\langle A, X \rangle, \langle B, Y \rangle) \vdash x \in \Delta$	<b>specs</b> $EQ \cup \{f = \langle g, h \rangle\}$
– $\Gamma, g: \mathcal{C}(A, B), h: \mathcal{D}(X, Y) \vdash x \in \Delta$	<b>specs</b> $EQ$

**Fig. 3.** Calculus for witness construction

To find a set of first-order specification equations that satisfies a given type judgment, our witness construction strategy iteratively applies rules of the calculus of witness construction until specification equations can be constructed and then composes the specification equations at the leave nodes of the decomposition tree into specification equations for the main goal. If more than one rule can be applied, it applies them in a predefined order of “simplicity”. If a rule generates more than one subgoal sequent, then the solution for the first subgoal may be used while solving the second.

We have integrated this strategy into a proof tactic `ProveIso` for proving isomorphisms between categories whose proofs are considered trivial by mathematicians. `ProveIso` first unfolds the definition of isomorphisms and decomposes the proof goal. Afterwards the witness construction strategy will guess values for the functors  $\theta$  and  $\eta$  between the two categories and finally the automated proof search procedure `AutoCAT2` will be called to validate that  $\theta$  and  $\eta$  are indeed functors of the appropriate types and that they are inverse to each other.

```

*- PRF : Iso-curry-v2 @edd,ck @sem
* top
∀C,D,E:Categories. Fun[C×D,E] ≅ Fun[C, Fun[D,E]]
* BY ProveIso
* top
∀C,D,E:Categories. Fun[C×D,E] ≅ Fun[C, Fun[D,E]]
* BY UnravelStatement
* 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 GuessFunctors
* 1 1
4. theta : Fun[Fun[C×D,E], Fun[C, Fun[D,E]]]
5. θ1F1A1X ≡ F1<A, X>
  ^ θ1F1A2k ≡ F2<1A, k>
  ^ θ1F2f X ≡ F2<F, 1X>
  ^ θ2ϕ X X1 ≡ ϕ <X, X1>
6. eta : Fun[Fun[C, Fun[D,E]], Fun[C×D,E]]
7. η1F1<A, X> ≡ F1A1X
  ^ η1F2<F, g> ≡ ((F2f cod(g))◦F1dom(F)2g)
  ^ η2ϕ <A, X1> ≡ ϕ A X1
⊢ θ and η are inverse
* BY AutoCAT2
    
```

We have applied this tactic to a small collection of isomorphism problems involving functor categories, product categories, and opposite categories. In each case, the isomorphism could be proven by `ProveIso` without a need for further interaction with the user. The screenshot above shows the formal proof that the functor categories  $\text{Fun}[C \times D, E]$  and  $\text{Fun}[C, \text{Fun}[D, E]]$  are isomorphic. On the top-level of this proof a user will only see that the proof was successful (indicated by a star in front of the tactic call `BY ProveIso`). Most users will be satisfied with that amount of information. For users interested in details of the proof the Nuprl system can display the proof tree in several layers of abstraction. The first layer,

shown in the snapshot as well, reveals the key idea that was necessary to solve the problem, but hides the tedious details involved in validating the solution.

Users interested in even more details may subsequently unfold the complete proof tree. However, one should be aware that this tree is huge. It takes 1046 and, respectively, 875 basic inferences to prove that  $\theta$  and  $\eta$  are indeed functors of the appropriate types and another 1141 inferences to prove that they are inverse to each other. The overall structure of this proof is similar to the hand-constructed one described in [Koz04], which required many hours of careful work to complete. In contrast to that the creation of the proof with `ProveIso` was fully automated and took only a few seconds to complete.

## 5.2 Proving Categories to Be Naturally Isomorphic

Proving the naturality of an isomorphism between two categories  $C_1$  and  $C_2$  is more demanding than just proving them to be isomorphic since the inverse functors  $\theta$  and  $\eta$  between  $C_1$  and  $C_2$  now have to be natural transformation of type  $\text{Fun}[\text{CAT}, \text{Cat}](U, V)$  and  $\text{Fun}[\text{CAT}, \text{Cat}](V, U)$ , where  $\text{CAT}$  is a yet to be determined product of the large categories  $\text{Cat}$  and  $\text{Cat}^{\text{op}}$  that fit the component categories of  $C$  and  $D$  and their polarities<sup>3</sup> and  $U$  and  $V$  are (unknown) elements of  $\text{Fun}[\text{CAT}, \text{Cat}]$ .

Constructing  $\text{CAT}$  is straightforward. For each component category of  $C_1$  we determine the polarity of its occurrence in  $C_1$ , and choose the category  $\text{Cat}$  if it occurs positively in  $C_1$  (and  $C_2$ , respectively) and  $\text{Cat}^{\text{op}}$  if it occurs negatively. If the respective polarities are different in  $C_1$  and  $C_2$ , then there is no simple natural isomorphism between the two categories and construction fails.<sup>4</sup> For the isomorphism between  $\text{Fun}[C \times D, E]$  and  $\text{Fun}[C, \text{Fun}[D, E]]$ , for instance,  $\text{CAT}$  has to be  $\text{Cat}^{\text{op}} \times \text{Cat}^{\text{op}} \times \text{Cat}$ , since  $C$  and  $D$  occur with negative polarities in  $\text{Fun}[C \times D, E]$  and  $\text{Fun}[C, \text{Fun}[D, E]]$  while  $E$  occurs positively.

Finding the functors  $U$  and  $V$  seems more difficult but is still considered obvious because all the relevant information for specifying them is expected to be contained in the terms that describe the construction of  $C_1$  and  $C_2$  from their components. Applying the object component of  $U$  and  $V$  to a tuple of component categories, for instance, has to result in  $C_1$  and  $C_2$ , respectively. Therefore specifications for  $U^1$  and  $V^1$  and also a typing of  $U^2$  and  $V^2$  can be easily constructed and a procedure similar to our witness construction strategy should be able to find a complete specification of all the first-order components of  $U$  and  $V$ . For the natural isomorphism between  $\text{Fun}[C \times D, E]$  and  $\text{Fun}[C, \text{Fun}[D, E]]$  this approach gives us the two specifications

$$U^1(C, D, E) = \text{Fun}[C \times D, E] \quad \text{and} \quad V^1(C, D, E) = \text{Fun}[C, \text{Fun}[D, E]]$$

as well as the typings

<sup>3</sup> A negative polarity indicates that a component category occurs on the left side of a functor. Otherwise the polarity is positive.

<sup>4</sup> We believe that in such situations the two categories are not isomorphic at all.

$$U^2(f, g, h) \in \text{Fun}[\text{Fun}[C \times D, E], \text{Fun}[C' \times D', E']] \text{ and}$$

$$V^2(f, g, h) \in \text{Fun}[\text{Fun}[C, \text{Fun}[D, E]], \text{Fun}[C, \text{Fun}[D, E]]].$$

To solve the latter requirement for  $U$ , we have to construct a functor between the categories  $\text{Fun}[C \times D, E]$  and  $\text{Fun}[C' \times D', E']$  from functors  $f, g$ , and  $g$  between  $C$  and  $C'$ ,  $D$  and  $D'$ , and  $E$  and  $E'$ . This construction depends only on the construction of the category  $\text{Fun}[C \times D, E]$  from its components but not on the fact that  $\text{Fun}[C \times D, E]$  and  $\text{Fun}[C, \text{Fun}[D, E]]$  have to be isomorphic. Since  $U$  shall be the domain of a natural transformation on large categories, one should expect that there is a “natural” method to construct its arrow component in a way that fits the construction of its object component.

Therefore, our approach to finding specifications that satisfy the requirements on the functors  $U$  and  $V$  is based on the hypothesis that there should be a natural method to extend a function that constructs a category from component categories into a functor on the category of categories whose object component is the given function. To formalize such a method we have developed a *calculus of constructor functors*. In this calculus we consider constructs like product, coproduct, functor, opposite, empty, or unit categories as object component of a functor on large categories and describe an arrow component that is naturally associated with it. For example, the *product constructor* is described as functor  $\mathcal{F}_{\text{Prod}} : \text{Fun}[\text{Cat} \times \text{Cat}, \text{Cat}]$ , where  $\mathcal{F}_{\text{Prod}}^1(C, D) = C \times D$  and  $\mathcal{F}_{\text{Prod}}^2(f, g) = (f, g)$ . The *functor constructor* is a functor  $\mathcal{F}_{\text{Fun}} : \text{Fun}[\text{Cat} \times \text{Cat}, \text{Cat}]$ , where  $\mathcal{F}_{\text{Fun}}^1(C, D) = \text{Fun}[C, D]$  and  $\mathcal{F}_{\text{Fun}}^2(f, g)^1(F) = g \circ F \circ f$   $\mathcal{F}_{\text{Fun}}^2(f, g)^2(\varphi) = g^2 \circ \varphi \circ f^1$ .

Using the specifications of elementary constructor functors a constructor for a given category is constructed by decomposing the category into basic constructor functions and composing the associated functors accordingly.

We have integrated this strategy into a proof tactic `ProveNatIso` for proving isomorphisms between categories to be natural. Like `ProveIso`, `ProveNatIso` first unfolds and decomposes the definition of natural isomorphisms. Then the domain and codomain  $U$  and  $V$  of the natural transformations will be constructed using the calculus of constructor functors and afterwards the natural transformations  $\theta$  and  $\eta$  using the calculus of witness construction. Finally the tactic `AutoCAT2` will validate the required properties on  $\theta$  and  $\eta$ .

We have applied this tactic to the same collection of isomorphism problems as before and were able to prove all isomorphisms to be natural. The screenshot below shows the formal proof that the functor categories  $\text{Fun}[C \times D, E]$  and  $\text{Fun}[C, \text{Fun}[D, E]]$  are naturally isomorphic as well as the first layer of the proof tree that has been constructed by `ProveNatIso`.

Again, the overall structure of this proof is similar to the hand-constructed one described in [Koz04], which indicates that the strategy does indeed automate the most obvious line of reasoning. Furthermore, the fact that the proofs of all “trivial” isomorphisms could be proven without a need for further interaction with the user shows that these proofs are in fact trivial in the sense that only natural constructions, standard forms of reasoning, and meticulous attention to detail are required to solve the problem.

```

*- PRF : NatIso-curry @edd,ck @sem
* top
Fun[C<D,E] and Fun[C, Fun[D,E]] are naturally isomorphic
* BY ProveNatIso.
NatIso-curry 2012_01_25-AM-08_16_33 @edd,ck @sem
* top 1
∃U, V: Fun[Cat-op<Cat-op<Cat, Cat]
  ∃θ: Fun[Cat-op<Cat-op<Cat, Cat] (U, V),
  ∃η: Fun[Cat-op<Cat-op<Cat, Cat] (V, U),
  ∀C, D, E: Cat,
    Fun[C<D,E] ≅ Fun[C, Fun[D,E]]
    via θ <C, D, E> and η <C, D, E>

* BY InstConstructors
* 1 1
1. U : Fun[Cat-op<Cat-op<Cat, Cat]
2. U1 <C, D, E> ≅ Fun[C<D,E]
  ^ U2 <P, Q, R>1 F1 <A, X> ≅ R1 F1 <P1A, Q1X>
  ^ U2 <P, Q, R>1 F2 <f, g> ≅ R2 F2 <P2f, Q2g>
  ^ U2 <P, Q, R>2 φ <A, X> ≅ R2 φ <P1A, Q1X>
3. V : Fun[Cat-op<Cat-op<Cat, Cat]
4. V1 <C, D, E> ≅ Fun[C, Fun[D,E]]
  ^ V2 <P, Q, R>1 F1 A1 B ≅ R1 F1 P1 A1 Q1 B
  ^ V2 <P, Q, R>1 F1 A2 h ≅ R2 F1 P1 A2 Q2 h
  ^ V2 <P, Q, R>1 F2 k Y ≅ R2 F2 P2 k Q1 Y
  ^ V2 <P, Q, R>2 φ X Y ≅ R2 φ P1 X Q1 Y
  ⊢ ∃θ: Fun[Cat-op<Cat-op<Cat, Cat] (U, V),
    ∃η: Fun[Cat-op<Cat-op<Cat, Cat] (V, U),
    ∀C, D, E: Cat,
      Fun[C<D,E] ≅ Fun[C, Fun[D,E]]
      via θ <C, D, E> and η <C, D, E>

* BY CreateIsoFunctionors
* 1 1 1
5. theta : Fun[Cat-op<Cat-op<Cat, Cat] (U, V)
6. θ <C, D, E>1 F1 A1 X ≅ F1 <A, X>
  ^ θ <C, D, E>1 F1 A2 k ≅ F2 <A, k>
  ^ θ <C, D, E>1 F2 f X ≅ F2 <f, X>
  ^ θ <C, D, E>2 φ X X1 ≅ φ <X, X1>
7. eta : Fun[Cat-op<Cat-op<Cat, Cat] (V, U)
8. η <C, D, E>1 F1 <A, X> ≅ F1 A1 X
  ^ η <C, D, E>1 F2 <f, g> ≅ ((F2 f cod(g)) ∘ F1 dom(f)2 g)
  ^ η <C, D, E>2 φ <A, X1> ≅ φ A X1
  ⊢ ∀C, D, E: Cat,
    Fun[C<D,E] ≅ Fun[C, Fun[D,E]]
    via θ <C, D, E> and η <C, D, E>

* BY AutoCAT2

```

## 6 Conclusion and Future Work

We have presented an implementation of Kozen’s axiomatization of elementary category theory [Koz04] using the Nuprl logical framework [AC<sup>+</sup>00, AB<sup>+</sup>05] and a collection of proof tactics that automate standard patterns of reasoning in logic and category theory. We have demonstrated the effectiveness of this approach by automatically deriving proofs of natural isomorphisms between categories, one example of which is presented in detail above. The system works very well on the examples we have tried.

There is a number of alternative approaches to a formalization and automation of category theory. The Mizar approach [Miz, Try92, Ba01a, Ba01b, Ba01c] aims at a formal reconstruction of mathematical knowledge in a computer-oriented environment. Mizar’s library seems to contain the most comprehensive collection

of theorems but it does not provide a mechanism for automating domain-specific reasoning tasks.

The system of C accamo and Winskel [CW01] presents a second order calculus for a fragment of category theory, which permits a more elegant representation of higher-order constructs like functors or natural transformations. Since higher-order reasoning is more difficult to automate, however, facts like Yoneda’s lemma need to be stated as a rules instead of being derived as theorems.

Burstall and Rydeheard [RB88] have implemented a substantial fragment of *Computational Category Theory* in Standard ML. The focus of their work, however, was not on the development of proofs but on creating a basis for the use of category theory in program design.

Other approaches aim at formalizations of category theory in interactive proof systems. Glimming [Gli01] describes a development of basic category theory and a couple of concrete categories (Unit, Set, Gal, Cat, Poset and Cpo) in Isabelle/HOL. O’Keefe [O’K04] presents a formalization of category theory in Isabelle that focuses on the readability of proofs, aiming at a representation close to one in a mathematical textbook. In both approaches there are no attempts to improve automation beyond Isabelle’s generic prover.

Dyckhoff [Dyc08] presents an formalization of category theory in Martin-L of type theory that has some similarity to Kozen’s first-order axiomatization but uses higher-order constructs in some of the rules. Dyckhoff hints at techniques to automate reasoning in his calculus but there is no actual implementation.

In the Coq library there are two contributions concerning category theory. The development of Sa ibi and Huet [HS95, Sai95] contains definitions and constructions up to cartesian closed categories, which are then applied to the category of sets. The formalization is directly based on the Coq’s type theory. In contrast to that Simpson’s formalization [Sim04] is set up in a ZFC-like environment and includes some tactics to improve the automation. Both approaches, however, are not described in any official publication.

A key difference between these works and our approach is that we have given a full implementation of an independent calculus for reasoning about category. In addition, we have provided a family of tactics that allow many proofs to be automated. None of the other 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.

There are a number of technical insights that we have observed in the course of this work, which show the advantage of using the Nuprl system as framework for implementing category theory.

- The use of abstractions and display forms is 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 readability.
- The combination of rule objects and rule compiler are essential for a faithful implementation of proof calculi. Rule objects contain visual representations of proof rules that look almost identical to the version on paper and can

easily be checked for correctness while the actual implementation of the rule is being generated by the rule compiler.

- Formal proofs, even of elementary facts, have thousands of basic inferences, which are often quite tedious and do not lend much insight. This indicates to us that elementary category theory is a very good candidate for automation.
- The ability to inspect proof objects at increasing levels of abstraction makes it possible to generate proofs that humans can understand and check to the very last detail even if the formal proof is extremely large.
- Nuprl’s tactic mechanism makes it possible to quickly implement and test new reasoning strategies and to embed new reasoning patterns when they are discovered in the course of a not yet fully automated proof.
- Reasoning in elementary category theory can be automated very well once there is an understanding of the typical kinds of reasoning that mathematicians consider obvious. As most reasoning steps are based on straightforward decomposition and directed rewriting for equations, proof strategies spend most of their time building the proof. Apart from guessing witnesses, which involves investigating a small set of alternative choices, there is virtually no backtracking involved and the bulk of the development is completely deterministic, being driven by typing considerations.
- Proofs that are considered trivial from an intellectual point of view are in fact trivial in the sense that a computer program can find them without having to rely on sophisticated heuristics.

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 automate proofs that require witnesses for existential quantifiers. We believe that our *calculus for witness construction* will be useful beyond proofs of isomorphisms, as the most obvious solution for a problem in category theory is often the “simplest” element of the given type, which is exactly what the strategy generates. In the same way our *calculus of constructor functors* should be useful beyond proofs of natural isomorphisms, as it provides functors and natural transformations that come naturally with a given category.

To improve both the efficiency of a proof search and the readability of the constructed proofs we also plan to introduce higher levels of reasoning that compose theorems about general category-theoretical arguments instead of only applying basic inference rules. This form of compositional reasoning has proven successful in the formal optimization of communication systems [LK<sup>+</sup>99], where we could reduce a huge amount of basic inference steps to a proof that could be constructed in a few seconds.

Finally, we would like to mention an intriguing theoretical open problem. The proofs of natural isomorphisms between two categories  $C_1$  and  $C_2$  that we have described break down into two parts. The first part argues that  $C_1$  and  $C_2$  are isomorphic, and the second part argues that the isomorphism is natural. As Mac Lane describes it [McL71, 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 re-parameterizing the result in terms of functors in place of objects, natural transformations in place of arrows. But any constructions in the formal proof  $\pi$  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  $\pi$  is similarly parameterized. In fact, our proof strategy based on the calculus of constructor functors has attempted exactly that and succeeded in proving an isomorphism to be natural in each case where we could prove two categories to be isomorphic.

This leads us to ask: Under what conditions can one extract a proof of naturality *automatically* from  $\pi$ ? 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 somewhat analogous to the extraction of programs from proofs according to the Curry–Howard isomorphism. We believe that extracting naturality is possible at least for categories that can be described in terms of constructor functors, as this leads immediately to the domains and codomains of the natural transformations  $\theta$  and  $\eta$  between the two categories while  $\theta$  and  $\eta$  are constructed in the same way as in the proof the isomorphism between  $C_1$  and  $C_2$ . But a formal proof of this conjecture still needs to be given.

**Acknowledgements.** We thank Dexter Kozen for introducing us to the topic of implementing and automating elementary category theory and for explaining the informal reasoning patterns behind the proof in [Koz04].

## References

- [AB<sup>+</sup>05] Allen, S., Bickford, M., Constable, R., Eaton, R., Kreitz, C., Lorigo, L., Moran, E.: Innovations in computational type theory using Nuprl. *Journal of Applied Logic* 4(4), 428–469 (2006)
- [AC<sup>+</sup>00] Allen, S., Constable, R., Eaton, R., Kreitz, C., Lorigo, L.: The Nuprl Open Logical Environment. In: McAllester, D. (ed.) *CADE 2000*. LNCS, vol. 1831, pp. 428–469. Springer, Heidelberg (2000)
- [Ba01a] Bancerek, G.: Concrete categories. *Journal of Formalized Mathematics* 13 (2001)
- [Ba01b] Bancerek, G.: Miscellaneous facts about functors. *Journal of Formalized Mathematics* 13 (2001)
- [Ba01c] Bancerek, G.: Categorical background for duality theory. *Journal of Formalized Mathematics* 13 (2001)
- [BW90] Barr, M., Wells, C.: *Category Theory for Computing Science*. Prentice Hall (1990)
- [CA<sup>+</sup>86] Constable, R., et al.: *Implementing Mathematics with the Nuprl proof development system*. Prentice Hall (1986)
- [Con08] Constable, R.: Computational Type Theory. *Scholarpedia* 4(2), 7618 (2008)

- [CW01] C accamo, M.J., Winskel, G.: A higher-order calculus for categories. Technical Report RS-01-27, BRICS, University of Aarhus (2001)
- [Dyc08] Dyckhoff, R.: Category theory as an extension of Martin-L of type theory. Research Report CS/85/3, Revised 1988 (1988)
- [EM45] Eilenberg, S., MacLane, S.: General theory of natural equivalences. *Trans. Amer. Math. Soc.* 58, 231–244 (1945)
- [Gli01] Glimming, J.: Logic and automation for algebra of programming. Master thesis, University of Oxford (2001)
- [HS95] Huet, G., Sa ibi, A.: Constructive category theory. In: Joint CLICS-TYPES Workshop on Categories and Type Theory. MIT Press (1995)
- [EB70] Knuth, D., Bendix, P.: Simple word problems in universal algebra. In: *Computational Problems in Abstract Algebra*, pp. 263–297. Pergamon Press (1970)
- [KHH98] Kreitz, C., Hayden, M., Hickey, J.: A Proof Environment for the Development of Group Communication Systems. In: Kirchner, C., Kirchner, H. (eds.) *CADE 1998. LNCS (LNAI)*, vol. 1421, pp. 317–331. Springer, Heidelberg (1998)
- [Koz04] Kozen, D.: Toward the automation of category theory. Technical Report 2004-1964, Computer Science Department, Cornell University (2004)
- [KKR06] Kozen, D., Kreitz, C., Richter, E.: Automating Proofs in Category Theory. In: Furbach, U., Shankar, N. (eds.) *IJCAR 2006. LNCS (LNAI)*, vol. 4130, pp. 392–407. Springer, Heidelberg (2006)
- [Kre02] Kreitz, C.: The Nuprl Proof Development System, V5: Reference Manual and User’s Guide. Computer Science Department, Cornell University (2002)
- [Kre04] Kreitz, C.: Building reliable, high-performance networks with the Nuprl proof development system. *Journal of Functional Programming* 14(1), 21–68 (2004)
- [LK<sup>+</sup>99] Liu, X., Kreitz, C., van Renesse, R., Hickey, J., Hayden, M., Birman, K., Constable, R.: Building reliable, high-performance communication systems from components. In: *17th ACM Symposium on Operating Systems Principles*, vol. 34(5), pp. 80–92 (1999); *Operating Systems Review*
- [McL71] MacLane, S.: *Categories for the Working Mathematician*. Springer (1971)
- [Miz] Mizar home page, <http://www.mizar.org>
- [ML84] Martin-L of, P.: *Intuitionistic Type Theory*, Bibliopolis (1984)
- [O’K04] O’Keefe, G.: Towards a readable formalisation of category theory. In: Atkinson, M. (ed.) *Computing: The Australasian Theory Symposium. ENTCS*, vol. 91, pp. 212–228. Elsevier (2004)
- [RB88] Rydeheard, D., Burstall, R.: *Computational Category Theory*. International Series in Computer Science. Prentice Hall (1988)
- [RRW91] Reed, G.M., Roscoe, A.W., Wachter, R.F.: *Topology and Category Theory in Computer Science*. Oxford University Press (1991)
- [Sai95] Sa ibi, A.: *Constructive category theory* (1995), <http://coq.inria.fr/contribs/category.tar.gz>
- [Sim04] Simpson, C.: *Category theory in ZFC* (2004), <http://coq.inria.fr/contribs/CatsInZFC.tar.gz>
- [Try92] Trybulec, A.: Some isomorphisms between functor categories. *Journal of Formalized Mathematics* 4 (1992)