Natural Typesetting of Naproche Formalizations in LaTeX

Tim Lichtnau[1], Jonas Lippert[1] and Peter Koepke[1]

[1]University of Bonn, Endenicher Allee 60, 53115 Bonn, Germany

Abstract

We present two formalizations with the natural proof assistant ℕaproche which explore the notational and typographical potential of the new LaTeX input format. The first formalization leads up to Yoneda's Lemma in category theory. Commutative diagrams are defined and printed using the LaTeX package `tikz-cd`. A specific ℕaproche module translates a commutative diagram defined by `tikz-cd` commands into equalities of functional compositions which are proved by the ℕaproche reasoner. The second formalization is an abstract version of Euclid's proof of the infinitude of primes, using general algebraic concepts. It uses LaTeX commands to follow the common mathematical practice of making some obvious symbols implicit and hide them from the typeset output. The hidden information can be revealed by hovering over the PDF output.

Adequate typographical representations are crucial for the communication and understanding of mathematics. LaTeX is universally used for mathematical typesetting; diagrams convey geometric intuitions of complex situations; on the other hand some information needs to be hidden so that central ideas come to the fore. Although special notation is indispensable for mathematical writing, and although it is generally well-understood by experts, there are hardly any fixed rules. The choice of notation is usually at the author's discretion and may be a matter of conventions and styles. The two formalizations presented in this paper have been the topic of a Bachelor thesis project by the second author [1] and a seminar project by the first author [2] with special emphases on commutative diagrams and information hiding, respectively.

The natural proof assistant ℕaproche ("Natural Proof Checking") accepts input in the controlled natural language ForTheL ("Formula Theory Language"), approximating ordinary mathematical language and texts, including symbolic material [3]. The latest version of ℕaproche also introduces a LaTeX dialect for ForTheL so that high-quality mathematical typesetting is readily available. ℕaproche is available in the Isabelle Prover IDE, together with example formalizations from various domains of university mathematics in a style that is naturally readable by mathematicians [4]. ForTheL documents in LaTeX format use the file extension `.ftl.tex` which is continuously checked for logical correctness by the ℕaproche component of the Isabelle platform. At the same time the file can be typeset and printed by LaTeX. This means that a `.ftl.tex` file can be projected out semantically by a parsing process to (first-order) symbolic logic, while it can also be projected to mathematical typesetting. Some LaTeX commands like the theorem environment

14th Conference on Intelligent Computer Mathematics, July 26–31, 2021, Timisoara, Romania
s6tilich@uni-bonn.de (T. Lichtnau); jlippert@uni-bonn.de (J. Lippert); koepke@math.uni-bonn.de (P. Koepke)
0000-0002-2266-134X (P. Koepke)

```
\begin{theorem} ... \end{theorem}
```

are interpreted logically *and* typographically. Other LaTeX commands are ignored by the logical processing. One can also arrange that parts of the logical content are *not* printed out to reduce typographical complexity and to enhance understanding. This interplay is a topic of ongoing development and research.

Our first formalization explores the treatment of commutative diagrams in category theory. Diagrams are input linearly by `tikz-cd` commands. These can be printed in two dimensions using `tikz-cd`. To extract the logical content of a diagram, a new (sub-)parser for `tikz-cd` commands was written which finds all paths through a finite diagram and then generates function identities for distinct paths whose sources and targets agree. The use and treatment of diagrams is demonstrated in the axiomatic setup of categories and in the proof of the Yoneda lemma.

The second formalization presents an "algebraic" approach to arithmetic and primes. In ForTheL, the distributivity of a ring R can be expressed by:

```
\begin{Axiom}[DistribI]
Let $x,y,z \in \sR{R}$.
$x \tR{R} (y \pR{R} z) = (x \tR{R} y) \pR{R} (x \tR{R} z)$.
\end{Axiom}
```

where `\sR{R}`, `\sR{R}`, and `\sR{R}` stand for the underlying set, the multiplication, and the addition of R resp. Often a structure is identified with its underlying set, and it is clear from the context which multiplication and addition are to be used. Such simplifications facilitate reading and understanding of texts and can be implemented by "forgetful" LaTeX functions:

```
\newcommand{\sR}[1]{#1}
\newcommand{\tR}[1]{\cdot}
\newcommand{\pR}[1]{\cdot}
```

The print-out of the above distributive law thus takes the familiar form

Axiom. (DistribI) Let $x, y, z \in R$. $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$.

without extra indices like $\cdot_R$ or $\cdot^R$ etc. Note that we output the formal parts of ForTheL texts on a light-gray background throughout this paper.

Such "hacks" lead to a conveniently readable Euclid formalization and motivate further systematic work on presenting or hiding information in the LaTeX projection. This is also related to elaboration mechanisms like in Lean, where implicit variables can be hidden since their values can be derived automatically [5].

The apparent loss of information in the PDF output can be counteracted by dynamic features of electronic documents. In our formalization we employ a hovering mechanism to display extra information in a balloon, which is, e.g., available for Adobe Reader and some other PDF viewers. Augmenting the definitions of the LaTeX commands `\sR{R}`, `\tR{R}`, and `\pR{R}` we can arrange that complete terms and possibly other information are displayed on "mouse-over".

1. Commutative Diagrams and the Yoneda Lemma in Category Theory

Commutative diagrams are essential in category theory. Natural formalizations require pretty-printing of diagrams by some LaTeX package and the logical checking of their mathematical content. For this purpose, we have added an experimental module to Naproche which reads out functional equalities from commutative diagrams defined in the `tikz-cd` environment. We have used this in a Naproche formalization of the beginnings of category theory up to the Yoneda-Lemma. The complete formalization is available at [1]. The introductory chapter of the thesis contains information how to run Naproche with the diagram module.

As an example, we give the definition of the one-sorted notion of a category as we use it in our formalization. Standard categories can be viewed as one-sorted by identifying every object with the identity function on that object itself (see [1]).

Definition. (Category) A category is a collection of arrows C such that
(for every arrow f such that $f \in C$ we have
$s[f] \in C$ and $t[f] \in C$ and $t[s[f]] = s[f]$ and $s[t[f]] = t[f]$ and
....
....
and (for each arrow f, g such that $f, g \in C$ we have
$(t[f] = s[g] \implies$ (there is an arrow h such that $h \in C$ and

$$\begin{array}{ccc} s[f] & \xrightarrow{f} & t[f] \\ & \overset{h}{\searrow} & \downarrow{\scriptstyle g} \\ & & t[g] \end{array}$$

and for every arrow k such that $k \in C$ and $g \circ_c f = k$ we have $h = k$)))
and for all arrows f, g, h such that $f, g, h \in C$ and $t[f] = s[g]$ and $t[g] = s[h]$

$$\begin{array}{ccc} s[f] & \xrightarrow{f} & t[f] \\ {\scriptstyle (g \circ_c f)}\downarrow & & \downarrow{\scriptstyle (h \circ_c g)} \\ t[g] & \xrightarrow{h} & t[h] \end{array} .$$

We briefly describe the implementation of the diagram parser and its use. For a more detailed description see [1].

Our new Haskell module provides a translation from `tikz-cd` code to a conjunction of equations which is then translated to FOL by the ForTheL parser for further checking.

```
tikzcdP :: String -> String
tikzcdP a = maybe " Error: No proper diagram found."
  (printEq . mkEqtClasses . arrows . mkPaths . snd) (runParser diagramP a)
```

If no error is thrown, this function

1. identifies the arrows of a diagram through `diagramP`
2. lists all possible paths with `mkPaths`
3. finds paths that are meant to be equal by `mkEqtClasses`
4. conjuncts these equations via `printEq`

The input is parsed into a `Diagram` type, which is a list of `Arrow`. An `Arrow` is a triple consisting of a name, source and target. A position is a tuple of integers as we view diagrams as two dimensional objects on a grid. We start in the upper left corner on `initpos = (0,0)` and count down each row in the left entry and up each column on the right. Since we want to identify paths with the same source and target, we define a corresponding equality on `Arrow` and obtain an instance of `Eq`.

```
data Arrow = Arrow
  { name   :: String
  , source :: Position
  , target :: Position
  } deriving Show

instance Eq Arrow where
  (==) a b = source a == source b && target a == target b

data Diagram = Diagram
  { arrows :: [Arrow]
  } deriving Show

diagramP :: Parser Diagram
diagramP = Diagram <$> many arP

arP :: Parser Arrow
arP = Parser $ \input -> do
  let pos = pstn input
  (input1, rest)  <- runParser (spanP (/= '{')) input
  let newpos = src rest pos
  (input2, _)    <- runParser (charP '{') input1
  (input3, dr)   <- runParser (spanP (/= '}')) input2
  (input4, _)    <- runParser (spanP (/= '{')) input3
  (input5, _)    <- runParser (charP '{') input4
  (input6, name) <- runParser (spanP (/= '}')) input5
  (input7, _)    <- runParser (charP '}') input6
  case name /= " " of
    True  -> Just (show newpos ++ input7, Arrow name newpos (trgt dr newpos))
    _     -> Nothing
```

The diagram parser repeats `arP` as often as possible. The arrow parser waits for braces to get the direction and name of the current arrow. The current position is added to the beginning of the unparsed input at `show newpos ++ input8` to keep track of it. When the next arrow is parsed, we get this position by `pstn input`.

All possible paths are built by:

```
mkPaths1 :: Arrow -> Arrow -> [Arrow]
mkPaths1 a x
  | source x == target a  =
        [Arrow ((name x) ++ "  \\mcirc " ++ (name a)) (source a) (target x)]
  | source a == target x  =
        [Arrow ((name a) ++ "  \\mcirc " ++ (name x)) (source x) (target a)]
  | otherwise             = []

mkPaths2 :: Arrow -> Diagram -> Diagram
mkPaths2 x d = case arrows d of
  [] -> Diagram []
  (y:ys) -> Diagram $ mkPaths1 x y
            ++ arrows (mkPaths2 x (Diagram ys))

mkPaths :: Diagram -> Diagram
mkPaths d = case arrows d of
  [] -> Diagram []
  x : xs -> Diagram $ x : arrows (mkPaths (Diagram (xs ++ newA)))
    where
      newA = arrows $ mkPaths2 x $ Diagram xs
```

The thesis [1] contains a correctness proof for this algorithm.

Now the path complete diagram has to be partitioned by our predefined equality relation for `Arrow`. For an arrow `x` and a list of arrows `xs`, `mkEqtClass2` compares any arrow of `xs` with `x` and creates a list of arrows equal to `x`. Finally, `x` itself is added.

```
mkEqtClass1 :: (Eq a) => a -> a -> [a]
mkEqtClass1 x y
  | x == y = [y]
  | otherwise = []

mkEqtClass2 :: (Eq a) => a -> [a] -> [a]
mkEqtClass2 x xs = case xs of
  [] -> [x]
  y : ys -> mkEqtClass2 x ys ++ mkEqtClass1 x y
```

Then `mkEqtClasses` creates a class `z` by taking the first arrow `y` from a list of arrows `xs` and comparing it to the rest `ys`. The recursive call of this function is restricted to the original list minus the generated class. (This behaves well since partitions are disjunct.)

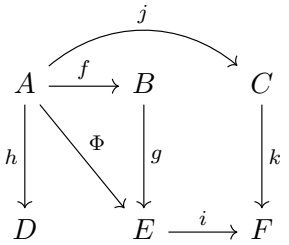
```
mkEqtClasses :: (Eq a) => [a] -> [[a]]
mkEqtClasses xs = case xs of
  [] -> []
  y : ys -> z : mkEqtClasses (ys \\ z)
    where z = (mkEqtClass2 y ys)
```

Finally, the classes are printed out by connecting class members with \= and classes with and.

```
printEq1 :: [Arrow] -> String
printEq1 l = case l of
  [] -> " "
  [x] -> name x
  x : xs -> name x ++ "  = " ++ printEq1 xs

printEq :: [[Arrow]] -> String
printEq a = case filter (\l -> length l >= 2) a of
  [] -> " "
  [l] -> printEq1 l -- ++ " ."
  l : ls -> printEq1 l ++ "  and " ++ printEq ls
```

To test the parser consider the following diagram:



The parser yields

```
Just
  ( " (-2,1) & F"
  , Diagram
    { arrows =
      [ Arrow {name = " f", source = (0,0), target = (0,1)}
      , Arrow {name = " h", source = (0,0), target = (-2,0)}
      , Arrow {name = " \\Phi", source = (0,0), target = (-2,1)}
      , Arrow {name = " j", source = (0,0), target = (0,2)}
      , Arrow {name = " g", source = (0,1), target = (-2,1)}
      , Arrow {name = " k", source = (0,2), target = (-2,2)}
      , Arrow {name = " i", source = (-2,1), target = (-2,2)}]})
```

The final output is

```
\Phi = g \mcirc f and i \mcirc \Phi = i \mcirc g \mcirc f = k \mcirc j
```

where \mcirc is the composition function of arrows in our formalization of categories.

We want to demonstrate the usage of commutative diagrams within our formalized proof of the Yoneda-Lemma. Note that $\mathrm{Nat}[C, D, F, G]$ is the class of all natural transformations $\alpha : F \to G$ where $F, G : C \rightrightarrows D$. Such a natural transformation has its d-component $T[C, D, F, G, \alpha, d]$ for every $d \,\epsilon\, C$.

Axiom. (PhiDef) Let F be a functor from C to SET. Let $c \,\epsilon\, C$. Let $\alpha \in Nat[C, SET, HomF[C, c], F]$.

$$\Phi(\alpha) = T[C, SET, HomF[C, c], F, \alpha, s[c]](s[c]).$$

Lemma. (Yoneda) Let C be a locally small category. Let F be a functor from C to SET. Let $c \,\epsilon\, C$ and $s[c] = c$.

$$\Phi \text{ is a bijection between } Nat[C, SET, HomF[C, c], F] \text{ and } F[c].$$

For comparison we give a textbook version as it can be found in [6].

Lemma. (Yoneda) For any functor $F : C \to Set$, whose domain C is locally small, and any object $c \in C$, there is a bijection $\mathrm{Nat}(C(c, -), F) \cong Fc$ that associates a natural transformation $\alpha : C(c, -) \Rightarrow F$ to the element $\alpha_c(1_c) \in Fc$.

Let us compare the diagrams of a standard proof of the Yoneda-Lemma with the diagrams in our single-sorted $\mathbb{N}$aproche formalization. In the second diagram we have $s[g]$ and $t[g]$ corresponding to the objects d and e on the left as source and target of g.

Proof. The association $\Phi(\alpha) \coloneqq \alpha_c(1_c)$ is clearly well defined. We need to verify for any element $x \in F(c)$, that $\Psi(x)$, defined componentwise by $\Psi(x)_d(f) \coloneqq F(f)(x)$ for each $d \in C$ and $f : c \to d$, is the inverse of Φ. Any element $f : c \to d$ of $C(c, d)$ is sent to $F(f)$ applied on $\Phi(\alpha) = x$.

By naturality of α,

$$\begin{array}{ccc} C(c,c) & \xrightarrow{\alpha_c} & F(c) \\ {\scriptstyle f_*}\downarrow & & \downarrow{\scriptstyle F(f)} \\ C(c,d) & \xrightarrow[\alpha_d]{} & F(d) \end{array}$$

commutes.

For all arrows d, f such that $d \,\epsilon\, C$ and $f \,\epsilon\, Hom[C, c, d]$ we have

$$\begin{array}{ccc} Hom[C,c,c] & \xrightarrow{T[C,SET,HomF[C,c],F,\alpha,c]} & F[c] \\ {\scriptstyle HomF[C,c][f]}\downarrow & & \downarrow{\scriptstyle F[f]} \\ Hom[C,c,d] & \xrightarrow[T[C,SET,HomF[C,c],F,\alpha,d].]{} & F[d] \end{array}$$

Hence

$$\Psi(\Phi(\alpha))_d(f) = F(f)(\Phi(\alpha)) = F(f)(\alpha_c(1_c)) = \alpha_d(f_*(1_c)) = \alpha_d(f).$$

Further we have

$$\Phi(\Psi(x)) = \Psi(x)_c(1_c) = F(1_c)(x) = 1_{F(c)}(x) = x.$$

To see why $\Psi(x)$ is a natural transformation we have to show that

$$\begin{array}{ccc} C(c,d) & \xrightarrow{\Psi(x)_d} & F(d) \\ {\scriptstyle g_*}\downarrow & & \downarrow{\scriptstyle F(g)} \\ C(c,e) & \xrightarrow[\Psi(x)_e]{} & F(e) \end{array}$$

commutes for every arrow $g \in C(d,e)$.

Let us show that for any arrow g such that $g \in C$ we have

$$\begin{array}{ccc} Hom[C,c,s[g]] & \xrightarrow{T[C,SET,HomF[C,c],F,\Psi(x),s[g]]} & F[s[g]] \\ {\scriptstyle HomF[C,c][g]}\downarrow & & \downarrow{\scriptstyle F[g]} \\ Hom[C,c,t[g]] & \xrightarrow[T[C,SET,HomF[C,c],F,\Psi(x),t[g]].]{} & F[t[g]] \end{array}$$

By definition, $\Psi(x)_d$ sends $f : c \to d$ to $F(f)(x)$ and $\Psi(x)_e$ sends $g_*(f)$ to $F(g_*(f))(x)$. Therefore, the claim follows by functoriality of F. □

Diagrams are an indispensable notation for equal paths in categories. They illustrate the mathematical core and make it more comprehensible for the reader. We have reproduced the proof above in ℕaproche using the diagram package.

Our notation of natural transformations in ℕaproche mentions all parameters explicitly, which makes the display somewhat bulky. The next section discusses methods to hide obvious or trivial parameters. We plan on combining those methods with the diagram parser so that that the final diagram might take the form

$$\begin{array}{ccc} C(c,d) & \xrightarrow{\Psi(x)_d} & F[d] \\ {\scriptstyle g_*}\downarrow & & \downarrow{\scriptstyle F[g]} \\ C(c,e) & \xrightarrow[\Psi(x)_e]{} & F[e] \end{array}$$

Note that on certain PDF viewers hovering over the blue parts will reveal the terms with all parameters.

2. Hiding Implicit Notation in a Proof of Euclid's Theorem

We present an approach to the infinitude of primes via wellfounded order relations in rings, defined by divisibility. The complete formalization can be found in [2]. Prime numbers are minimal elements in the divisibility order and Euclid's main argument is captured abstractly by the order theoretic *Stranger Theorem*:

Theorem. Let O be a wellfounded order. Assume for every element x of O there exists a $y \in O$ such that x and y have no common predecessors by O. Then

$$\{m \in O \mid m \text{ is a minimum of } O\}$$

has no upper bound by O.

We work in general structures like magmas, monoids or rings to exhibit the abstract arguments. Although ℕaproche does not yet have dedicated mechanisms for algebraic structures, we can make use of the first-order approach to typing to deal with subtyping and coercions.

We introduce a method to only display "mathematically relevant" information in the LaTeX-output to prevent losing the overview. The hidden information is not completely lost in the document, since it can be made visible by moving the mouse over the text. Thus uncertainty about meanings of terms can be resolved without inspecting the `.ftl.tex` file.

2.1. (Un)hiding informations in Magmas

A magma is a set together with a binary operation on it. In principle we must distinguish between a magma M and the underlying set $|M|$ since there may be two different magma structures on the same set. We first introduce the new word magma by the signature-command

Signature. A magma is a notion.

and then we pretype some variable to be a magma:

Let M denote a magma.

Its underlying set can be defined as

```
\begin{signature}
 $\sGC{M}$ is a set.
\end{signature}
```

The function `\sGC` has two faces: from the ℕaproche perspective the function `sGC` is a class function from the class of magmas to the class of sets; from the LaTeX perspective, the function `sGC` puts absolute value bars around its argument, as it is defined by

```
\newcommand{\sGC}[1]{\vert #1 \vert}
```

The binary operation on a magma can be introduced by

```
\begin{signature}
 Let $x,y \in \sGC{M}$. $x \gdot{M} y$ is an element of $\sGC{M}$.
\end{signature}
```

Note that the operation depends on the parameter M, so that `\gdot` really is a *ternary* mixfix function. The logical processing by ℕaproche requires all three parameters to determine the magma under consideration. But if the magma is understood from the context one usually suppresses that parameter in the printed output. This can be achieved by defining the LaTeX command `\gdot` as

```
\newcommand{\gdot}[1]{\cdot}
```

Omitting obvious parameters of a function is not the only situation in which we may use such "hacks". One often identifies a structure with its underlying set. So we want to print out M instead of $|M|$. This can be achieved by redefining the LaTeX command `\sGC` without the vertical bars.

A reader may be interested to see the full information behind the LaTeX printout. This is possible without looking up the `.ftl.tex`-file. Some PDF viewers, such as Adobe Reader or

Foxit Reader, feature so-called tooltips for editable PDF files to show alternative information in a box through a mouseover event. This allows to define a version of `\sGC` which normally omits the vertical bars, but shows the bars on mouseover, using the LaTeX command `\tooltip`:

```
\newcommand{\sG}[1]{\mbox{\tooltip{$#1$}{$|#1|$}}}
```

For ℕaproche, `\sG` should have the same meaning as `\sGC`. This can be arranged by an alias command of the form

Let M stand for $|M|$.

Similarly for the operation on the magma we can define:

```
\newcommand{\gdot}[1]{\mbox{\tooltip{$\cdot$}{$\cdot_{#1}$}}}
```

The associative law then becomes:

Definition. M is associative iff $(x{\cdot}y){\cdot}z = x{\cdot}(y{\cdot}z)$ for all $x, y, z \in M$.

Everything which is colored blue is linked to a hidden expression which gets visible if one moves the mouse over it. Note, that the color of the multiplication symbol is difficult to see.

2.2. Monoids

Now we expand the magma-theory to monoids.

Definition. A monoid is an associative magma with a neutral element.

For eventual study of prime numbers we define divisibility in a Monoid M:

Definition. Let $x, y \in M$. x divides y in M iff there exists $k \in M$ such that $k{\cdot}x = y$.

Now we already can prove the transitivity of divisibility :

Lemma. (transitiveDiv) Let $k, m, n \in M$. Suppose n divides m in M and m divides k in M. Then n divides k in M.

Proof. Take an $l \in M$ such that $l{\cdot}n = m$.
Take an $p \in M$ such that $p{\cdot}m = k$.
Then $k = p{\cdot}m = p \cdot(l{\cdot}n) = (p{\cdot}l){\cdot}n$. Thus n divides k in M. □

2.3. The divisor order-relation

Divisibility defines an order. As with magmas we introduce orders O together with their underlying set $|O|$ and appropriate tooltips and aliases. In ℕaproche, relations can be declared with the keyword “atom”.

Signature. Let $x, y \in O$. xOy is an atom.

Definition. Let $x \in O$. A predecessor of x by O is an element y of O such that $y{\leq}x$.

To see the implementation of partial orders, minima and upper bounds, see [2].
Now we combine the theory of monoids and orders to obtain the divisor-relation on a monoid.

Let M denote a monoid.

Definition. $|$ is an order on M such that for any $x, y \in M$ we have $x|y$ iff x divides y in M .

Lemma. $|$ is reflexive and transitive.

Proof. $|$ is reflexive. Indeed x divides x in M for all $x \in M$.
$|$ is transitive (by transitiveDiv). □

Since our goal is to show that there are infinitely many primes, we may want to use a property of finiteness: If you have finitely many elements $a_1, \ldots, a_n$ of a monoid you can multiply them all together. This is captured order-theoretically as follows:

Axiom. (FiniteMultiplication) Let M be an abelian Monoid. Let X be a finite subset of M. Then X has an upper bound by $|$.

The further formalization of groups (i.e. the definition of inverse elements in monoids) can be found in [2].

2.4. Rings

A ring has an (additive) abelian group structure and a (multiplicative) monoid structure both of which have the same underlying set. If R denotes a ring and $|R|$ is the underlying set, we can introduce the extra structure as follows:

Signature. $\mathrm{Ab}(R)$ is an abelian group based on R.

Signature. $\mathrm{Mu}(R)$ is a Monoid based on R.

Let $x{+}y$ stand for $x \cdot_{\mathrm{Ab}(R)} y$.
Let $x{\cdot}y$ stand for $x \cdot_{\mathrm{Mu}(R)} y$.

0 and 1 can be defined as the neutral elements of $\mathrm{Ab}(R)$ and $\mathrm{Mu}(R)$ resp. Adding distributitivity axioms, we have completely formalized the algebraic structure of rings.

2.5. Wellfounded Orders

We now introduce an order-theoretic concept which will give strong results about minimal elements (which later will be standard prime numbers).

Definition. O is wellfounded iff O is a partial order and for all nonempty subsets S of $|O|$ there exists a minimum of S with O.

Definition. $\mathcal{M}(O) = \{m \in O \mid m \text{ is a minimum of } O\}$.

To show the strength of the concept of wellfounded orders, here is one example:

Lemma. Let O be a wellfounded order. Let $x \in O$. Then there exists a predecessor y of x by O such that y is a minimum of O.

Proof. Define $X = \{z \in O \mid z \leq x\}$. $X \subseteq O$. Take a minimum z of X with O. z is a minimum of O. □

The idea of relative primality is captured by the notion of *stranger*.

Definition. Let $x \in O$. A stranger of x in O is an element y of O such that x and y have no common predecessors by O.

Theorem. (StrangerTheorem) Let O be a wellfounded order. Assume every element of O has a stranger in O. Then $\mathcal{M}(O)$ has no upper bound by O.

Somewhat surprisingly, Naproche could find a derivation without an explicit proof given.

2.6. The ring of integers

We now introduce $\mathbb{Z}$ as a commutative ring and

Signature. $\mathbb{N}_{>0}$ is a submonoid of $\mathrm{Mu}(\mathbb{Z})$.

We call the elements of $\mathbb{N}_{>0}$ positive numbers. This signature command requires, that $\mathbb{N}_{>0}$ is closed under the $\mathbb{Z}$-multiplication and that 1 is a positive number.
Furthermore we require for positive number n, m:

Axiom. (MultEquiv) Assume n divides m in $\mathbb{Z}$ and m divides n in $\mathbb{Z}$. Then $n = m$.

Note, that we used another tooltip here. If you don't use the long-term-expressions provided by tooltip you have to be careful, since in the PDF file a blue $\mathbb{Z}$ can indicate either the underlying set or the underlying monoid. It helps the reader, that the context ('$\in \mathbb{Z}$' or 'divides ... in $\mathbb{Z}$') already forces a disambiguation.
The key point of the last axiom is, that $|$ is a partial order.

Axiom. $n+m$ is a nontrivial positive number,

where nontrivial means $\neq 1$.
We denote the order $|_{>1}$ for $|$ restricted to the set of nontrivial positive numbers $\mathbb{N}_{>1}$. The reason to use this restriction of the underlying set is that prime numbers are exactly the minimal elements of $|_{>1}$ (i.e. a nontrivial positive number such that the only nontrivial divisor is itself).
We need one final axiom:

Axiom. Let S be a nonempty subset of $\mathbb{N}_{>1}$. Then there exists a minimum of S with $|$.

Of course $|_{>1}$ is a partial order, since it is the restriction of the partial order $|$. The axiom ensures, that $|_{>1}$ is also wellfounded.

2.7. Euclids Theorem

Lemma. (ExistenceOfStrangers) Every element of $\mathbb{N}_{>1}$ has a stranger in $|_{>1}$.

Indeed, $x+1$ is a stranger of x in $|_{>1}$. This follows, since a nontrivial common divisor of x and $x+1$ would also divide the difference 1 (lemma divDif [2]) but is also divisible by 1. By (MultEquiv) it already has to be trivial.

Letting $\mathbb{P}$ denote the set of prime numbers, we get Euclid's theorem in $\mathbb{N}$aproche:

Theorem. (Euclid) $\mathbb{P}$ is not finite.

Proof. Proof by contradiction. Assume $\mathbb{P}$ is finite.
$\mathbb{P} = \mathcal{M}(|_{>1})$.
$\mathcal{M}(|_{>1})$ has no upper bound by $|_{>1}$ (by ExistenceOfStrangers, StrangerTheorem).
$\mathcal{M}(|_{>1})$ is a finite subset of $\mathbb{N}_{>0}$. $\mathbb{N}_{>0}$ is an abelian monoid.
Take an upper bound b of $\mathcal{M}(|_{>1})$ by $|$ (by FiniteMultiplication).
b = 1. $\mathcal{M}(|_{>1})$ is nonempty.
Contradiction. □

References

[1] Lippert, Jonas, Natürliche Formalisierung der Kategorientheorie, https://github.com/jonaslippert/Bachelor-thesis/blob/main/Arbeit/Arbeit.pdf, 2021.

[2] Lichtnau, Tim, Euclids Theorem, https://github.com/naproche/FLib/tree/master/NumberTheory, 2021.

[3] A. D. Lon, P. Koepke, A. Lorenzen, Interpreting Mathematical Texts in Naproche-SAD, in: C. Benzmüller, B. Miller (Eds.), Intelligent Computer Mathematics. CICM 2020, volume 12236 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 284–289.

[4] The Isabelle Proof Assistant, 2021. URL: https://isabelle.in.tum.de/.

[5] L. M. de Moura, J. Avigad, S. Kong, C. Roux, Elaboration in dependent type theory, CoRR abs/1505.04324 (2015). URL: http://arxiv.org/abs/1505.04324. arXiv:1505.04324.

[6] E. Riehl, Category theory in context, Dover Publications, Mineola, New York, 2016.