Metacat
a categorical framework for formal systems

Paul Wilson
Hellas AI
paul@hellas.ai
paul@statusfailed.com

We present a categorical framework for formal systems in which inference rules with m metavariables over a category of syntax $\mathscr{S}$, taken to be a cartesian PROP, are represented by operations of arity $k \to n$ equipped with spans $k \leftarrow m \to n$ in $\mathscr{S}$, encoding the hypotheses and conclusions in a common metavariable context. Composition is by substitution of metavariables, which is the sole primitive operation, as in Metamath.

Proofs in this setting form a symmetric monoidal category whose monoidal structure encodes the combination and reuse of hypotheses. This structure admits a proof-checking algorithm; we provide an open-source implementation together with a surface syntax for defining formal systems. As a demonstration, we encode the formulae and inference rules of first-order logic in Metacat, and give axioms and representative derivations as examples.

1 Introduction

Metacat is a framework for presenting formal systems in a way that makes the compositional structure of proofs explicit. Inference rules are modeled as transformations of *input syntax* to *output syntax* by substitution of metavariables. The resulting proofs are not merely derivation trees written in a different notation: they assemble into a symmetric monoidal category, so that independent subproofs can be juxtaposed and reused in a principled way. This categorical structure gives a semantics for proof assembly and suggests a natural visual language in terms of string diagrams. For example, a proof of $\vdash P \Rightarrow P$ using the Metamath axioms ax-1 (Simp), ax-2 (Frege), and ax-mp (modus ponens) is depicted as a string diagram below.

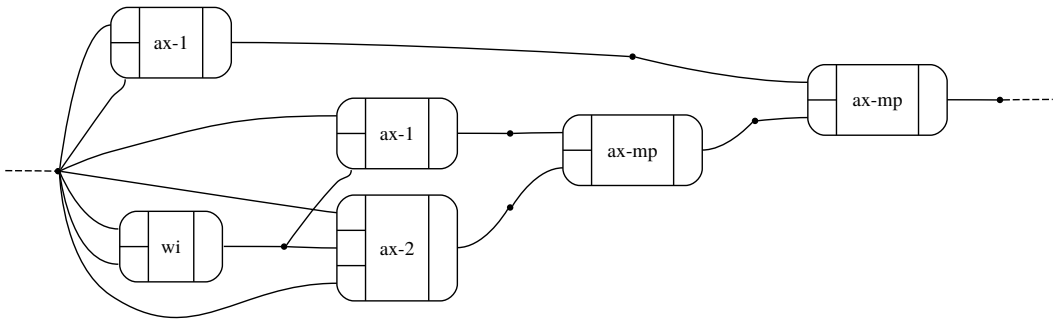


Figure 1: A string-diagrammatic proof of $\vdash P \Rightarrow P$.

This perspective is motivated by desire for a small, core proof kernel. In Metamath and Metamath Zero for example, the essential operation is substitution of formulae for metavariables, with very little

Submitted to:
ACT 2026

logic built into the kernel itself. We adopt that discipline here. A minimal substitution-based kernel is attractive for two reasons. First, as the Metamath Zero thesis makes clear [5], it is easier to trust and verify a small checker than a large proof engine with many primitive notions baked in. Second, keeping the kernel agnostic about a particular logic makes the framework flexible: the ambient machinery does not presuppose classical logic, type theory, or any other foundational choice, but instead lets a concrete formal system be specified by its syntax and rules. The categorical presentation clarifies this idea by treating substitution as composition and by treating collections of hypotheses as monoidal structure.

The categorical viewpoint also addresses two practical issues that arise in formal syntax. On the proof side, it provides formally justified tools for visualisation rather than relying on ad hoc proof drawings. On the syntax side, it allows for the combinatorial representation of proofs as cospans of hypergraphs [1, 2, 3], in which connectivity is primary and concerns about variable names or α-equivalence can be handled structurally. Our goal is therefore not to design a diagrammatic notation for first-order logic itself, in the style of systems where the diagrams are the formulas[4]. Rather, we use categorical and string-diagrammatic methods to expose the structure of proofs in a general substitutional framework.

The main contribution of this paper is a categorical account of formal proofs in which proof checking is reduced to evaluation in a symmetric monoidal category of partial functions. Proof rules are represented syntactically by spans over a category of syntax, derivations form a symmetric monoidal category $\mathscr{P}$, and there is an operational interpretation as partial functions

$$[\![-]\!] : \mathscr{P} \to \mathbf{Pfn}$$

such that a derivation is valid exactly when the corresponding partial map is defined at the claimed boundary. This interpretation is realized by Metacat, an implementation of the proof checker which verifies proofs by evaluating the associated open hypergraph.

1.1 Related Work

Metacat is closest in spirit to Metamath and Metamath Zero [10, 5]. It shares their emphasis on a tiny trusted core and on explicit substitution, but recasts that discipline in a language where compositional structure is first-class. One way to view the project is as a string-diagrammatic analogue of Metamath Zero: not a competing foundation, but a reformulation that makes the algebra of proofs easier to see and manipulate with the tools of category theory. The difference is one of presentation: rather than treating proofs primarily as trees of rule applications, we make their compositional structure explicit in a symmetric monoidal setting. In this respect the project also draws on categorical work on open graphs, hypergraphs, and diagrammatic rewriting [1, 2, 3], which provide natural combinatorial models for syntax with sharing. By contrast with systems such as Chyp [9], our proofs are not obtained by rewriting string diagrams. Rather, derivations are represented explicitly as diagrams and checked by an external operational semantics. It also contrasts with proof assistants based on richer built-in logics, such as Lean and Agda [12, 13], where considerably more logical structure is part of the trusted core. At the same time, our aim differs from diagrammatic logics such as diagrammatic first-order logic [4]: the diagrams here are not the formulas themselves, but rather a representation of the syntax and composition of proofs. Finally, unlike proof-net style representations in linear logic [8, 6, 11], we do not try to build global correctness directly into the proof syntax; as in Metamath, the syntax of derivations is simple, and correctness is imposed by a separate checking semantics.

1.2 Synopsis

The remainder of the paper is structured as follows. Section 2 introduces the categories of syntax and proofs used throughout the paper, and Section 3 gives the corresponding proof-checking semantics and operational interpretation. Section 4 then illustrates the framework on a fragment of first-order logic in the style of Metamath. Finally, Section 5 closes with brief remarks on implementation and future work.

2 Syntax and Proof

This section introduces the basic technical objects used throughout the paper. We begin with a motivating example from propositional logic, then explain how syntax with metavariables is represented categorically, and finally describe how axioms and inference rules generate a category of proofs over this syntax. The main point is that the interface of a proof already has symmetric monoidal shape: a proof takes a finite family of hypotheses to a finite family of conclusions, and larger proofs are built by juxtaposing and composing smaller ones.

2.1 A motivating example

Consider the inference rule of modus ponens. Informally, it consumes two hypotheses, namely a proposition A and an implication $A \Rightarrow B$, and produces the conclusion B. As a morphism in a symmetric monoidal category, we might think of it as having the type $A \otimes (A \Rightarrow B) \to B$. This already illustrates two features that will persist in the general theory. First, inference rules are naturally many-input, many-output operations. Second, they are parameterised by *metavariables*: the two occurrences of A and the two occurrences of B are not independent pieces of syntax, but must be understood as referring to the same placeholders throughout the rule.

This dependence on shared metavariables is exactly where naive representations of syntax become awkward. If formulas are represented only as strings or parse trees with named variables, then composition of rules requires bookkeeping for substitution, matching of repeated variables, and avoidance of accidental name capture. Those are familiar problems, but they obscure the compositional content we wish to expose. Our aim is instead to use a representation in which sharing is explicit and structural.

2.2 Categories of syntax

We begin by explicitly formalising the notion of syntax used in this paper.

Definition 2.1. *A* category of syntax $\mathscr{S}$ *is a free cartesian PROP generated by a set of operations* Σ_1.

Here Σ_1 should be thought of as the collection of term formers of the language. An operation $n \to 1$ in Σ_1 is therefore an n-ary term former. In propositional logic, for example, one may take a binary operation $\Rightarrow : 2 \to 1$ for implication and a unary operation $\neg : 1 \to 1$ for negation as elements of Σ_1. A map

$$f : m \to 1$$

is then interpreted as a term with m metavariables, or equivalently as a tree with m ‘holes’. More generally, a map

$$f : m \to n$$

is an n-tuple of such terms in a common metavariable context. Under this viewpoint, maps $0 \to 1$ correspond to object-level terms having no metavariables, while *generating* maps of type $0 \to 1$ are precisely the constants of the language.

For example, a composite like

$$\Rightarrow \fatsemi \neg : 2 \to 1$$

therefore represents the term $\neg(\phi_0 \Rightarrow \phi_1)$, where ϕ_0 and ϕ_1 are thought of as *metavariables*.

Notice that this encoding of syntax replaces plain trees by a representation in which metavariables and their sharing are part of the syntax itself. For example, the term $\neg(\phi_0 \Rightarrow \phi_0)$ - in which ϕ_0 is shared - is encoded using the *cartesian* structure of the category by explicitly copying ϕ_0.

That $\mathscr{S}$ is *cartesian* is exactly what allows metavariables to be duplicated or discarded when building terms. In particular, repeated use of the same metavariable, as for example in the term $\neg(\phi_0 \Rightarrow \phi_0)$, is represented internally by the syntax rather than imposed as extra bookkeeping on variable names. This is also why the combinatorial representation of morphisms as cospans of hypergraphs is natural here: sharing is explicit in the syntax.

2.3 Categories of proofs

Equipped with a notion of syntax analogous to formal language, we now turn our attention to categories of proofs, corresponding to formal systems.

Definition 2.2. *A category of proofs $\mathscr{P}$ over syntax $\mathscr{S}$ is the PROP freely generated by operations Γ where each operation $f : a \to b$ in Γ is equipped with a span*

$$a \xleftarrow{\mathsf{src}(f)} m \xrightarrow{\mathsf{tgt}(f)} b$$

in the syntax category $\mathscr{S}$, where m is a natural number denoting the number of metavariables. We call the arrows *of $\mathscr{P}$* derivations.

To illustrate this definition, let us return to our earlier example, modus ponens, which we now assume to be an operation $\mathsf{ax\text{-}mp} : 2 \to 1$ in Γ. We would like to think of it informally as a rule of shape

$$A \otimes (A \Rightarrow B) \to B,$$

where A and B are metavariables. In this case, we have $m = 2$ with the source and target maps given below left and right, respectively.

$$\mathsf{src}(\mathsf{ax\text{-}mp}) : (\phi_0, \phi_1) \mapsto (\phi_0,\ \phi_0 \Rightarrow \phi_1) \qquad \mathsf{tgt}(\mathsf{ax\text{-}mp}) : (\phi_0, \phi_1) \mapsto \phi_1$$

Thus a proof-generating operation carries not only an arity $a \to b$ in the PROP $\mathscr{P}$, but also source and target maps in $\mathscr{S}$ describing its a hypotheses and b conclusions as tuples of expressions in a common metavariable context, that is, a span in $\mathscr{S}$. In the present case, $\mathsf{src}(\mathsf{ax\text{-}mp}) : 2 \to 2$ encodes the pair of hypotheses $(\phi_0,\ \phi_0 \Rightarrow \phi_1)$, while $\mathsf{tgt}(\mathsf{ax\text{-}mp}) : 2 \to 1$ encodes the conclusion ϕ_1.

Remark 2.3. *Note that because $\mathscr{S}$ has cartesian structure, we may alternatively regard the $m \to a$ source map as being an a-tuple of independent $m \to 1$ source maps. We choose the $m \to a$ representation here so that common subterms built from metavariables can be explicitly shared in the source map, but the two are equivalent.*

Finally, notice that proof terms in this definition are essentially 'untyped': it is trivial to compose two proof generators whose associated source and target maps in $\mathscr{S}$ do not match appropriately. This is intentional: the syntax exists to be *checked*, rather than to be a structural representation of a valid proof (as for example in the proof nets of linear logic [8, 6, 11]).

3 Proof Checking

We now turn our attention to *checking* proofs. The section is separated into two parts: the formal semantics of proof checking, and an algorithm for checking proofs.

In Section 2, we defined categories of proofs $\mathscr{P}$ whose arrows were *derivations*. However, a proof is a derivation of a particular statement: its boundary in syntax.

Definition 3.1. *Let $\mathscr{P}$ be a category of proofs over $\mathscr{S}$, let $d : a \to b$ be an arrow in $\mathscr{P}$, and let $s : m \to a$ and $t : m \to b$ be arrows in $\mathscr{S}$. We say that (s,t) is a* type *for d.*

An arrow of $\mathscr{P}$ alone is only a derivation schema. To regard it as a proof, one must also specify which a hypotheses and which b conclusions in $\mathscr{S}$ it is intended to relate.

Types (s,t) play a role analogous to the source and target maps attached to the generators of $\mathscr{P}$, but we do not require a derivation to have a unique type[1]. This reflects the fact that a single derivation pattern may witness many different proof instances. For example, the metamath rule wn states that the negation of a well-formed formula is well-formed. Its generating source and target maps send ϕ_0 to $\mathsf{wff}(\phi_0)$ and $\mathsf{wff}(\neg(\phi_0))$, respectively. But the same derivation also serves as a proof of the more specific judgement taking $\mathsf{wff}(\neg(\phi_0))$ to $\mathsf{wff}(\neg(\neg(\phi_0)))$.

3.1 Valid and Invalid Proofs

So far, we have not said when a derivation $d : a \to b$ is a *proof* of a chosen type (s,t). The point of proof checking is precisely to decide this.

The idea is to map each generator f of a derivation into a partial map associated to the composite

$$\mathsf{src}(f)^- \fatsemi \mathsf{tgt}(f)^+,$$

where $\mathsf{src}(f)^-$ is a *pattern matcher*, deconstructing the input syntax to recover metavariables, while $\mathsf{tgt}(f)^+$ constructs the desired output syntax. This map is undefined precisely when the input syntax does not conform to the shape specified by $\mathsf{src}(f)$, meaning that the required hypotheses were not satisfied. Since derivations are built inductively from generators, checking a derivation d satisfies a type (s,t) is done by recursively interpreting generators in the above way, pre- and post-composing with s^+ and t^-, respectively, and evaluating the resulting map.

One can package this as an operational semantics. From the syntax category $\mathscr{S}$ one first constructs an auxiliary symmetric monoidal category which records both term formation and term matching.

Definition 3.2. *Let $\mathscr{S}$ be a category of syntax with generators Σ_1. Then $\mathscr{S}^\pm$ is the symmetric monoidal category presented by:*

- *a special Frobenius structure*

$$\mu : 2 \to 1, \qquad \eta : 0 \to 1, \qquad \delta : 1 \to 2, \qquad \varepsilon : 1 \to 0$$

- *for each generator $g : m \to n$ of $\mathscr{S}$, a* constructor *$g^+ : m \to n$ and naturality equations making g^+ a comonoid homomorphism with respect to δ and ε;*
- *for each generator $g : m \to n$ of $\mathscr{S}$, a* matcher *$g^- : n \to m$, and dual naturality equations making g^- a monoid homomorphism with respect to μ and η.*

[1] We suspect that any derivation admits a maximally general type, but do not pursue that question here.

Thus $\mathscr{S}^{\pm}$ contains two formal copies of the syntax-generating operations: one copy for building terms and one for deconstructing them. By Fox's algebraic presentation [7] of cartesian categories, the constructor side therefore carries a cartesian copy of $\mathscr{S}$ inside $\mathscr{S}^{\pm}$, with copying and discarding represented by δ and ε. Dually, the matcher side carries an opposite copy of $\mathscr{S}$, with the structural maps represented by μ and η. The existence of these two embeddings are formally stated as follows.

Theorem 3.3. *There exist strict symmetric monoidal functors*

$$(-)^{+} : \mathscr{S} \to \mathscr{S}^{\pm} \qquad \textit{and} \qquad (-)^{-} : \mathscr{S}^{\mathrm{op}} \to \mathscr{S}^{\pm}$$

defined inductively on the generators of $\mathscr{S}$ as follows:

- *for each $g : m \to n$ in Σ_1, one has $(g)^{+} = g^{+}$ and $(g^{op})^{-} = g^{-}$;*
- *the cartesian copying and discarding maps of $\mathscr{S}$ are sent by $(-)^{+}$ to δ and ε, respectively;*
- *the opposites of the same maps are sent by $(-)^{-}$ to μ and η, respectively.*

Proof. Since $\mathscr{S}$ is the free cartesian PROP on Σ_1, Fox's theorem [7] allows us to regard it as the strict symmetric monoidal category generated by Σ_1 together with maps 'copy' $\Delta : 1 \to 2$ and 'discard' $! : 1 \to 0$, *commutative comonoid* equations, and *naturality* equations $f \fatsemi \Delta_n = \Delta_m \fatsemi (f \otimes f)$ and $f \fatsemi !_n = !_m$ for each $f : m \to n$ in $\mathscr{S}$.

Now define the identity-on-objects strict symmetric monoidal assignment $(-)^{+}$ inductively on generators as below, so that $(f \fatsemi g)^{+} = (f)^{+} \fatsemi (g)^{+}$ $(f \otimes g)^{+} = (f)^{+} \otimes (g)^{+}$ by definition.

$$(\mathsf{id})^{+} = \mathsf{id} \qquad (\sigma)^{+} = \sigma \qquad (g)^{+} = g^{+} \qquad (\Delta)^{+} = \delta \qquad (!)^{+} = \varepsilon$$

Symmetric monoidal functoriality follows directly from this inductive definition, but to ensure well-definedness we must ensure the commutative comonoid and naturality equations are preserved. The former follow from the Special Frobenius laws (see e.g. [1]) for δ and ε. The latter follow in two parts. Naturality of copy/discard with respect to generators is guaranteed by the conditions in Definition 3.2, and on identity and symmetry follows from the laws of Special Frobenius categories [14, Proposition D.2]. Moreover, naturality equations are closed under composition and tensor product, so by induction they hold for every arrow generated from these cases. Thus, the assignment of $(-)^{+}$ on arrows of $\mathscr{S}$ is well-defined, giving a strict symmetric monoidal functor.

The construction of $(-)^{-}$ is dual. A generating operation $g : m \to n$ in $\mathscr{S}$ is an arrow $g^{op} : n \to m$ of $\mathscr{S}^{op}$, which is sent to $g^{-} : n \to m$. The opposite of the cartesian comonoid structure is a commutative monoid with dual naturality equations, so $\Delta^{op} : 2 \to 1$ is sent to μ and $!^{op} : 0 \to 1$ is sent to η. Equations are preserved for the same reason as for $(-)^{+}$: the commutative monoid equations follow from the Frobenius structure, and the dual naturality equations are part of the definition of $\mathscr{S}^{\pm}$. Thus, we also have a strict symmetric monoidal functor $(-)^{-} : \mathscr{S}^{\mathrm{op}} \to \mathscr{S}^{\pm}$. □

The functor $(-)^{+}$ is the covariant embedding of syntax as term formation. The functor $(-)^{-}$ is the contravariant embedding of syntax as pattern matching. In particular, if $\Delta : 1 \to 2$ and $! : 1 \to 0$ are the copying and discarding maps in $\mathscr{S}$, then $\Delta^{+} = \delta$ and $!^{+} = \varepsilon$, while $\Delta^{-} = \mu$ and $!^{-} = \eta$. More generally, a syntax map $u : m \to n$ may therefore be read in two complementary ways: covariantly, as building an n-tuple of terms from m metavariables, and contravariantly, as a matching procedure that tries to recover m metavariables from an n-tuple of terms.

3.2 Operational Semantics of Derivations

To make this precise, we interpret arrows of $\mathscr{S}^{\pm}$ as partial functions on tuples of rooted syntax trees. Fix a set T_{Σ_1} of trees generated inductively by:

- leaves labelled by natural numbers, representing metavariables;
- for each generator $g : m \to n$ in Σ_1, nodes $(g, i; t_1, \ldots, t_m)$ for $1 \leq i \leq n$ and trees $t_1, \ldots, t_m \in T_{\Sigma_1}$.

Thus a node remembers not only the operation label g, but also which output port i of g it represents.

Definition 3.4. *The* operational semantics *of $\mathscr{S}^{\pm}$ is the inductively defined interpretation[2] of its generators as partial functions on tuples of trees:*

- *a Frobenius spider $m \to n$ with $m = 0$ creates a fresh leaf and returns n copies of it[3];*
- *a Frobenius spider $m \to n$ with $m > 0$ is defined exactly when all m inputs are equal, in which case it returns n copies of that common tree;*
- *for a constructor generator $g^+ : m \to n$, the output on $(t_1, \ldots, t_m)$ is the n-tuple*

$$((g, 1; t_1, \ldots, t_m), \ldots, (g, n; t_1, \ldots, t_m));$$

- *for a matcher generator $g^- : n \to m$, the input must be an n-tuple of trees of the form*

$$(g, 1; t_1, \ldots, t_m), \ldots, (g, n; t_1, \ldots, t_m)$$

with common children $t_1, \ldots, t_m$, and the output is then $(t_1, \ldots, t_m)$; otherwise the function is undefined.

The meaning of an arbitrary arrow of $\mathscr{S}^{\pm}$ is then obtained inductively from this generating data using composition and tensor product. The special Frobenius structure accounts for equality checking and copying, while the generators g^+ and g^- account for construction and deconstruction of syntax nodes. In particular, every syntax map $u : m \to n$ gives rise, via the two embeddings above, to two arrows

$$u^+ : m \to n \qquad \text{and} \qquad u^- : n \to m$$

in $\mathscr{S}^{\pm}$, and hence to two partial functions:

$$[\![u^+]\!] : T_{\Sigma_1}^m \rightharpoonup T_{\Sigma_1}^n \qquad \text{and} \qquad [\![u^-]\!] : T_{\Sigma_1}^n \rightharpoonup T_{\Sigma_1}^m.$$

The first builds output syntax from metavariable assignments, while the second attempts to match a tuple of syntax trees against the pattern described by u.

Definition 3.5. *The* operational semantics of derivations *is the symmetric monoidal functor*

$$[\![-]\!] : \mathscr{P} \to \mathbf{Pfn}$$

where **Pfn** *is the category of partial functions on tuples of trees, with arrows $m \to n$ given by partial functions $T_{\Sigma_1}^m \rightharpoonup T_{\Sigma_1}^n$, determined on proof generators $f : a \to b$ by*

$$[\![f]\!] := [\![\mathsf{src}(f)^- \fatsemi \mathsf{tgt}(f)^+]\!].$$

[2]This interpretation respects the special Frobenius equations, as well as the constructor and matcher naturality equations of Definition 3.2, and is therefore well-defined.

[3]This requires the Frobenius structure to be represented in normal form: each connected component must have a well-defined arity and coarity. Otherwise the zero-input case and positive-input case are not syntactically distinguished. In the implementation, this normal form is implicit in the cospan-of-hypergraphs representation.

Thus a proof generator f is checked by the partial map

$$[\![\mathsf{src}(f)^- \fatsemi \mathsf{tgt}(f)^+]\!],$$

and the validity criterion for a derivation is simply the following: a derivation d with proposed boundary (s,t) is valid precisely when

$$[\![s^+]\!] \fatsemi [\![d]\!] \fatsemi [\![t^-]\!]$$

is defined. Note that in a minor abuse of notation, we here used $[\![\cdot]\!]$ for both the functor from $\mathscr{S}^{\pm}$ as well as the functor from $\mathscr{P}$ in Definition 3.5.

Implementation

These ideas are realized in the Metacat implementation, which provides a surface syntax for defining formal systems together with a checker for the resulting derivations; the command-line tool can be installed with `cargo install metacat-cli`. Concretely, a derivation is compiled to an open hypergraph, and the partial-function semantics above is evaluated by a topological traversal of its operations. The source nodes are first initialized with leaf values, representing the metavariables supplied at the boundary. One then executes the operations in topological order[4], propagating tree values through the hypergraph. Constructor nodes build syntax trees, matcher nodes inspect and deconstruct them, and the Frobenius structure accounts for copying and equality tests. In this way, proof checking becomes an executable evaluation procedure.

4 First order logic

As a case study, we now demonstrate how first order logic can be encoded in Metacat. Our implementation is a direct port from the Metamath [10] standard library, but we only include a small number of illustrative axioms and propositions. We begin by defining the relevant fragment of the category of syntax. For the propositional fragment considered first, the term formers are the judgement constructor wff together with the connectives $\neg$ and $\Rightarrow$. In addition, we include the judgement symbol $\vdash$, which will later be used to express derivability. Thus the syntax is generated by the following symbols:

$$\mathsf{wff} : 1 \to 1, \qquad \vdash : 1 \to 1, \qquad \neg : 1 \to 1, \qquad \Rightarrow : 2 \to 1$$

Notice that there are two judgement symbols. The constructor wff packages a raw formula as a well-formed formula, while $\vdash$ indicates *provability*. Although we will not study ill-formed expressions in any detail, the well-formedness judgement provides a simple first example of how a Metamath style grammar is represented in the present framework.

4.1 Well-Formed Formulae

The first proof generators we introduce are purely *syntactic*. They express that negation preserves well-formedness, and that implication takes two well-formed formulae to a well-formed formula. In Metamath notation these are the rules usually written wn and wi.

$$\mathsf{wn} : (\phi_0 \mapsto \mathsf{wff}(\phi_0)) \to (\phi_0 \mapsto \mathsf{wff}(\neg\phi_0))$$

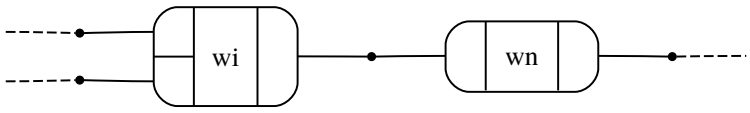
[4]Note that to order operations topologically, one must decompose a term's hypergraph structure into explicit spider operations, otherwise such an ordering may not exist.

$$\mathsf{wi} : (\phi_0, \phi_1 \mapsto (\mathsf{wff}(\phi_0), \mathsf{wff}(\phi_1))) \to (\phi_0, \phi_1 \mapsto \mathsf{wff}(\phi_0 \Rightarrow \phi_1))$$

More explicitly, wn is a proof generator of arity $1 \to 1$ whose source map sends a metavariable ϕ_0 to the single hypothesis $\mathsf{wff}(\phi_0)$ and whose target map sends ϕ_0 to the conclusion $\mathsf{wff}(\neg\phi_0)$. Similarly, wi is a proof generator of arity $2 \to 1$ whose source map sends (ϕ_0, ϕ_1) to the pair of hypotheses $(\mathsf{wff}(\phi_0), \mathsf{wff}(\phi_1))$ and whose target map sends the same metavariables to the conclusion $\mathsf{wff}(\phi_0 \Rightarrow \phi_1)$.

These two generators already illustrate the role of shared metavariables. The same metavariable assignment must be used consistently across both source and target maps: the occurrences of ϕ_0 and ϕ_1 in the conclusion of wi are not new variables, but the same placeholders that appeared in its hypotheses. This is exactly the kind of bookkeeping that is represented structurally by the syntax maps of Section 2.

Putting wn and wi together allows us to derive that the negation of an implication is also well-formed. Concretely, from hypotheses $\mathsf{wff}(\phi_0)$ and $\mathsf{wff}(\phi_1)$ one first uses wi to obtain $\mathsf{wff}(\phi_0 \Rightarrow \phi_1)$, and then applies wn to conclude $\mathsf{wff}(\neg(\phi_0 \Rightarrow \phi_1))$. The derivation of this statement is depicted below.



This example is deliberately elementary, but it shows the two key ingredients already at work: syntax maps describe the formulae mentioned by an inference rule, and proof generators compose exactly as derivations compose. Subsequent proof rules for propositional and first-order logic will build on the same pattern.

4.2 Propositional Logic

We can now encode some axioms of propositional logic, which will be sufficient to write the derivation in Figure 1. We now spell out the axioms and inference rules used; namely modus ponens together with ax-1 and ax-2:

$$\mathsf{ax\text{-}mp} : (\phi_0, \phi_1 \mapsto (\vdash(\phi_0), \vdash(\phi_0 \Rightarrow \phi_1))) \to (\phi_0, \phi_1 \mapsto \vdash(\phi_1))$$

$$\mathsf{ax\text{-}1} : (\phi_0, \phi_1 \mapsto (\mathsf{wff}(\phi_0), \mathsf{wff}(\phi_1))) \to (\phi_0, \phi_1 \mapsto \vdash(\phi_0 \Rightarrow (\phi_1 \Rightarrow \phi_0)))$$

$$\mathsf{ax\text{-}2} : (\phi_0, \phi_1, \phi_2 \mapsto (\mathsf{wff}(\phi_0), \mathsf{wff}(\phi_1), \mathsf{wff}(\phi_2))) \to (\phi_0, \phi_1, \phi_2 \mapsto \vdash((\phi_0 \Rightarrow (\phi_1 \Rightarrow \phi_2)) \Rightarrow ((\phi_0 \Rightarrow \phi_1) \Rightarrow (\phi_0 \Rightarrow \phi_2))))$$

As before, the source maps record the hypotheses required to form the rule, while the target maps record the proposition proved by that rule. Together with the well-formedness generators above, these proof generators are enough to express the derivation of the identity implication in Figure 1.

4.3 Quantifiers and First Order Logic

As a final comment, we must address the encoding of *quantifiers*. These present a particular challenge as *binders* of variables. In the present framework, the key point is that variables are not represented primarily by names, but by their incidence in the underlying syntax object. Accordingly, a quantified formula should not be thought of as a string together with a side condition on free and bound names. Rather, the variable bound by a quantifier is represented by explicit sharing in the syntax itself: the same metavariable is fed both into the quantifier node and into the formula being quantified.

For the fragment we consider here, we add a single constructor

$$\forall : 2 \to 1,$$

whose first input represents the bound variable and whose second input represents the body of the quantified formula. Diagrammatically, a formula such as $\forall x.\, \phi(x)$ is therefore encoded by a single shared input wire feeding both the variable slot of $\forall$ and the occurrences of that variable inside ϕ. This is exactly the sort of sharing that is awkward to express cleanly with a tree syntax plus named substitution, but is immediate in the present setting.

The first proof generator involving quantification is universal generalization:

$$\mathsf{ax\text{-}gen} : (x, \phi \mapsto\vdash (\phi)) \to (x, \phi \mapsto\vdash (\forall(x, \phi)))$$

This should be read with some care. The metavariable x is present in the common context of the rule, but it need not occur freely in the premise ϕ. That fact is represented structurally rather than by an external side condition: if the body ϕ ignores the variable input, then the syntax map for the premise simply discards it.

These examples illustrate the main point about binders in Metacat. The binding behaviour of $\forall$ is not enforced by a special substitution operation built into proofs; instead, it is already present in the syntax map describing the rule. In a fuller treatment one would distinguish variable sorts explicitly, as in Metamath's treatment of set variables, but we omit that additional structure here in order to keep the example focused on the categorical representation of sharing.

5 Conclusions and Future Work

We have presented a categorical framework for formal systems in which syntax is represented by a free cartesian PROP and proofs by a free PROP over that syntax. In this formulation, inference rules are many-input, many-output operations equipped with explicit source and target maps in syntax, and proof checking is separated from proof syntax itself. This makes it possible to represent invalid derivations syntactically while giving a precise checking semantics in terms of pattern matching and construction of syntax.

The symmetric monoidal structure is not an additional decoration on top of an existing proof formalism. Rather, it is the basic shape already present in proofs with multiple hypotheses and conclusions. Making that structure explicit gives a uniform language for proof assembly and opens the door to string-diagrammatic and combinatorial representations of formal reasoning.

We also described a small first-order case study in the style of Metamath, showing how well-formedness judgements, propositional axioms, and quantifier rules can be encoded in this framework.

Several directions remain open. The most immediate is tooling: definitions, abstraction mechanisms, and better proof authoring support are all needed if Metacat is to be used as a practical theorem-proving environment. In addition, we hope that the combinatorial nature of this proof representation can surface the inherent parallelism in proofs, and by leveraging data-parallel syntax representations like [14] provide automatic parallelisation of proof checking. On the theoretical side, it would be interesting to study richer logical fragments and to make more systematic use of categorical structure to relate different formal systems by translation and comparison.

References

[1] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Paweł Sobociński & Fabio Zanasi (2020): *String Diagram Rewrite Theory I: Rewriting with Frobenius Structure*. arXiv preprint arXiv:2012.01847.

[2] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Paweł Sobociński & Fabio Zanasi (2021): *String Diagram Rewrite Theory II: Rewriting with Monoidal Structure*. arXiv preprint arXiv:2104.14086.

[3] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Paweł Sobociński & Fabio Zanasi (2022): *String Diagram Rewrite Theory III: Confluence and Termination*. arXiv preprint arXiv:2201.XXXX.

[4] Filippo Bonchi, Alessandro Di Giorgio, Nathan Haydon & Pawel Sobocinski (2024): *Diagrammatic Algebra of First Order Logic*. arXiv:2401.07055.

[5] Mario Carneiro (2022): *Metamath Zero: From Logic to Proof Assistant to Verified Compilation*. Ph.D. thesis, Carnegie Mellon University. Available at `https://digama0.github.io/mm0/thesis.pdf`.

[6] Pierre-Louis Curien (2005): *Introduction to Linear Logic and Ludics, Part II*. arXiv preprint cs/0501039.

[7] Thomas Fox (1976): *Coalgebras and Cartesian Categories*. Communications in Algebra 4(7), pp. 665–667, doi:10.1080/00927877608822127.

[8] Jean-Yves Girard (1987): *Linear Logic*. Theoretical Computer Science 50, pp. 1–101. Available at `https://girard.perso.math.cnrs.fr/linear.pdf`.

[9] Aleks Kissinger (2023): *Chyp: An Interactive Theorem Prover for String Diagrams*. `https://github.com/akissinger/chyp`. Version 0.5.2, Apache-2.0 License.

[10] Norman Megill & David A. Wheeler (2019): *Metamath: A Computer Language for Mathematical Proofs*. Lulu Press. Available at `https://us.metamath.org/downloads/metamath.pdf`.

[11] Paul-André Melliès (2009): *Categorical Semantics of Linear Logic*. Lecture notes.

[12] Leonardo de Moura & Sebastian Ullrich (2021): *Lean 4: A Programming Language and Theorem Prover*. arXiv preprint arXiv:2104.XXXXX.

[13] Ulf Norell (2007): *Towards a Practical Programming Language Based on Dependent Type Theory*. In: International Conference on Types for Proofs and Programs (TYPES 2006), Springer, pp. 175–189.

[14] Paul Wilson & Fabio Zanasi (2023): *Data-Parallel Algorithms for String Diagrams*. arXiv:2305.01041.

A Labeled Identity Derivation

For reference, Figure 2 shows a version of the identity derivation from Figure 1, where nodes are labeled with a text representation of syntax maps using x_i for metavariables.

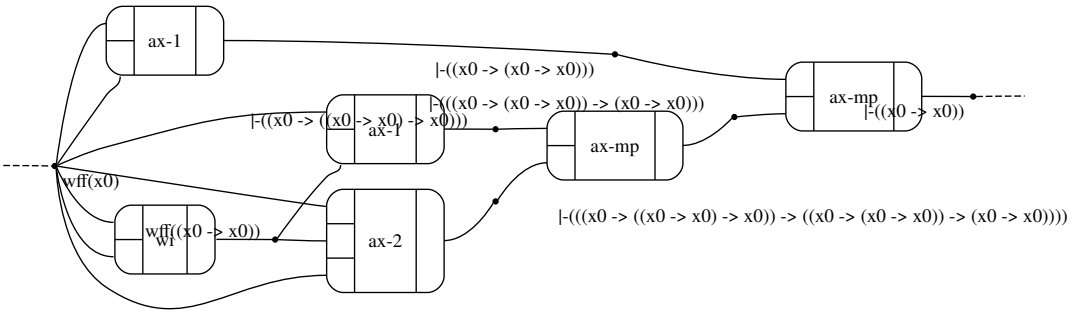


Figure 2: A version of the derivation of $\vdash P \Rightarrow P$ labeled with tree values at each node