The formalisation and computerization of languages of mathematics: The case for interleaving natural language with the formal language and how can this be computerised

Fairouz Kamareddine

`www.macs.hw.ac.uk/~fairouz/talks/talks2004/leipzig-tutorials04.pdf`*

3-4 May 2004

*Parts of this talk are based on Kamareddine [2001]; Kamareddine et al. [2002]; Kamaredine and Nederpelt [2004], and on joint work with Maarek and Wells in Kamaredine et al. [2004b,a]

A Century of Complexity

	1900	2000
Main way information travels in society:	paper	electric signals, radio
Number of parts in complex machine:	10,000 (locomotive)	1,000,000,000 (CPU)
Worst consequences of single machine failure:	100s die	end of all life?
Likelihood a machine includes a computer:	very low	very high

The Need for Formalism

- Because of the increasing interdependency of systems and the faster and more automatic travel of information, failures can have a wide impact. So *correctness* is important.

- Modern technological systems are just too complicated for humans to reason about unaided, so *automation* is needed.

- Systems have so many possible states that *testing* is often impractical. It seems that *proofs* are needed to cover infinitely many situations.

- So *some* kind of formalism is needed to *aid in design* and to *ensure safety*.

What Kind of Formalisms?

A reasoning formalism should *at least* be:

- *Correct*: Only correct statements can be "proven".
- *Adequate*: Needed properties in the problem domain can be stated and proved.
- *Feasible*: The resources (money, time) used in stating and proving needed properties must be within practical limits.

What Kind of Formalisms?

Assuming a minimally acceptable formalism, we would also like it to be:

- *Efficient*: Costs of both the reasoning process *and* the thing being reasoned about should be minimized.

- *Supportive of reuse*: Slight specification changes should not force reproving properties for an entire system. Libraries of pre-proved statements should be well supported.

- *Elegant*: The core of the reasoning formalism should be as simple as possible, to aid in reasoning about the formalism itself.

ULTRA Research Themes

Useful

Logics

Logic is the foundation for rigorous reasoning. There is an ongoing search for better logics and for better methods for verifying the correctness of logics.

Types

Types are a foundation for making logics more flexible without losing correctness and safety. Types are also being used increasingly often for analyzing complex higher-order systems.

Rewriting

Rewriting is using rules of logic, mathematics, or computation in a stepwise manner. Rewriting theory supports reasoning about equivalences between propositions or programs and efficient computation strategies.

and their

Automation

Modern theories of logic, types, and rewriting and the systems to which they are applied have become so complicated that automation is essential.

Applications

Systems of logic, types, and rewriting have applications in the design and implementation of programming languages and theorem provers, in mathematics and in natural language.

Proofs? Logics? What are they?

- A proof is the *guarantee* of some statement provided by a rigorous *explanation* stated using some *logic*.

- A logic is a formalism for statements and proofs of statements. A logic usually has *axioms* (statements "for free") and *rules* for combining already proven statements to prove more statements.

- For example, this is provable in the logic PROP:

$$A,\ B,\ A \rightarrow B \rightarrow C \vdash C$$

  This is not:

$$A,\ B,\ A \rightarrow D \rightarrow C \nvdash C$$

- Why do we believe the explanation of a proof? Because a proved statement is derived step by step from explicit assumptions using a trusted logic.

Logic is an Area of Active Research

- New logics are regularly invented for specialized purposes. Known logics may be *too inflexible* for the task. Or they may be *too flexible*, interfering with automated proof search.

- *Broken* logics are regularly invented. A recent example: The 1988 version of the OCL (Object Constraint Language) sublanguage of UML (Unified Modelling Language) had Russell's paradox of a nearly a century earlier! It is still not known if the revised OCL and/or UML is consistent.

- There has been an explosion of new logics in the 20th century. How do we know which ones to trust?

- Assume a problem Π,
  - If you *give* me an algorithm to solve Π, I can check whether this algorithm really solves Π.
  - But, if you ask me to *find* an algorithm to solve Π, I may go on forever trying but without success.

- But, this result was already found by Aristotle (384–322 B.C.) who wanted a set of rules that would be powerful enough for most intuitively valid proofs. Aristotle correctly stated that **proof search** is harder than **proof checking**:

  Assume a proposition Φ.
  - If you *give* me a proof of Φ, I can check whether this proof really proves Φ.
  - But, if you ask me to *find* a proof of Φ, I may go on forever trying but without success.

- Much later than Aristotle, Leibniz (1646–1717) conceived of **automated deduction**, i.e., to find

– a language L in which arbitrary concepts could be formulated, and
– a machine to determine the correctness of statements in L.

Such a machine can not work for every statement according to Aristotle and (later results by) Gödel and Turing.

Exercises

- Let b be the barber in village v who shaves all and only those men in v who do not shave themselves. Does b shave himself or does he not?

- Let R be the set of all and only those sets which do not contain themselves. Which statement is true: $R \in R$ or $R \notin R$?

- Let G be the set of all and only those finite games (i.e. games which end after a finite number of moves) between two players. Let hg (hypergame) be a game between two players which works as follows: player 1 chooses a game g from G, and then g is played between player 2 and player 1. I.e. player 1 chooses g, player 2 makes the first move, player 1 makes the second move and so on.
  - Show that hg is a finite game between two players, and hence include hg in G.
  - Show that G is no longer the set of finite games between two players.

- Take the sentence s to be: "I am not true". Is such a sentence true or false?

What are Types?

- Euclid's *Elements* (circa 325 B.C.) begins with:
  1. A *point* is that which has no part;
  2. A *line* is breadthless length.

     ⋮

  15. A *circle* is a plane figure contained by one line such that all the straight lines falling upon it from one point among those lying within the figure are equal to one another.

- Although the above seems to merely *define* points, lines, and circles, more importantly it *distinguishes* between them.

  This prevents undesired reasoning, like considering whether two points (instead of two lines) are parallel.

- *Undesired* reasoning? Euclid would have said: *impossible* reasoning. When considering whether objects are parallel, intuition implicitly forced Euclid to think about the *type* of the objects. Because intuition does not support parallel points, Euclid does not even *try* such reasoning.

Why Types are Needed for Logic

- Mathematical systems have become less intuitive, for several reasons:
  - very complex or abstract
  - formal
  - Something without intuition is using the system: a computer.
- Non-intuitive systems are vulnerable to *paradoxes*. The human brain's built-in type machinery can fail to warn against an impossible situation. Reasoning can proceed obtaining results that may be wrong or paradoxical.
- Example: Russell [1902] and Frege [1902] showed that Naive Set Theory had a *paradox*. Let S be "the set of all sets which do not contain themselves". Then, *both* of these are provable:

$$S \in S \qquad\qquad S \notin S$$

- Russell [1908] Russell began the use of *types* to solve this problem.

A Quick Introduction to Rewriting

We all know how to do *algebra*:

$$
\begin{array}{rlll}
 & \underline{(a+b)} - a & \text{by rule} & x + y = y + x \\
= & \underline{(b+a) - a} & \text{by rule} & x - y = x + (-y) \\
= & \underline{(b+a) + (-a)} & \text{by rule} & (x+y) + z = x + (y+z) \\
= & b + (\underline{a + (-a)}) & \text{by rule} & x + (-x) = 0 \\
= & \underline{b + 0} & \text{by rule} & x + 0 = x \\
= & b & &
\end{array}
$$

Rewriting is the action of replacing a subexpression which is matched by an instance of one side of a rule by the corresponding instance of the other side of the same rule. If you know algebra, you understand the basics of rewriting.

Important Issues in Rewriting

- *Orientation*: Usually, most rules can only be used from left to right as in $x + 0 \rightarrow x$. Forward use of the oriented rules represents progress in computation. Unoriented rules usually do trivial work as in $x + y = y + x$.

- *Termination*: It is desirable to show that rewriting halts, i.e., to avoid infinite sequences of the form $P \rightarrow P_1 \rightarrow P_2 \rightarrow \cdots$.

- *Confluence*: It is desirable that the result of rewriting is independent of the order in the rules used. For example, $1 + 2 + 3$ should rewrite to 6, no matter how we evaluate it.

The invention of computers and computability

- *Types* have *always existed* in mathematics, but not explicited until 1879. Euclid avoided *impossible* situations (e.g., two parallel points) via classes/*types*.
- In 19th century, controversies in analysis led to logical *precision*. (*Cauchy*, *Dedekind*, *Cantor*, *Peano*, *Frege*).
- In 1900, *Hilbert* posed an impressive list of difficult questions.
- One important question was: given a formula of predicate logic, can we decide whether the formula is true or false?
- It took more than 30 years to answer this is impossible: *Turing Machines*, *Goedel's incompleteness* and *Church's λ-calculus*.
- f is computable iff f can be computed on a Turing Machine.
- f is computable iff f can be definable in the λ-calculus.
- Types, Logics, and Rewriting have become the heart of Computer Science.

Higher-Order Rewriting and Logic

- Church's λ-calculus provides *higher-order* rewriting, allowing equations like:

$$f(\underline{(\lambda x.\, x + (1/x))5}) = f(5 + (\underline{1/5})) = f(\underline{5 + 0.2}) = f(5.2)$$

- Church [1940b] introduced the simply typed λ-calculus (STLC) and on top of it his Simple Type Theory (CSTT) to provide paradox-free logic. The modern descendant of CSTT is the so-called "higher-order logic" (HOL).

The Convergence of Logics, Types, and Rewriting

- Heyting [1934a], Kolmogorov [1932a], Curry and Feys [1958a] (improved by Howard [1980a]), and de Bruijn (Kamareddine et al. [2003]) all observed the "*propositions as types*" or "*proofs as terms*" (PAT) correspondence.

- In PAT, not only is the λ-calculus embedded in the *propositions* as in HOL, but the structure of *proofs* is also given by another level of λ-terms. λ-terms are viewed as proofs of the propositions represented by their types.

- Advantages of PAT include:
  - better proof manipulation,
  - better independent proof checking,
  - the extraction of computer programs from proofs, and
  - proving the consistency of the logic via the termination of the rewriting system.

The Goal: Open borders between mathematics, linguistics, logic and computation

- Ordinary mathematicians *avoid* formal mathematical logic.
- Ordinary mathematicians *avoid* proof checking (via a computer).
- Ordinary mathematicians *may use* a computer for computation: there are over 1 million people who use mathematica (including linguists, engineers, etc.).
- Mathematicians may also use other computer forms like Maple, Latex, etc.
- But we are not interested in only *libraries* or *computation* or *text editing*.
- We want *freedeom of movement* between mathematics, linguistics, logic and computation.
- At every stage, we must have *the choice* of the level of formalilty and the depth of computation.

Common Mathematical Language of mathematicians: CML

+ CML is *expressive*: it has linguistic categories like *proofs* and *theorems*.

+ CML has been refined by intensive use and is rooted in *long traditions*.

+ CML is *approved* by most mathematicians as a communication medium.

+ CML *accommodates many branches* of mathematics, and is adaptable to new ones.

– Since CML is based on natural language, it is *informal* and *ambiguous*.

– CML is *incomplete:* Much is left implicit, appealing to the reader's intuition.

– CML is *poorly organised:* In a CML text, many structural aspects are omitted.

– CML is *automation-unfriendly:* A CML text is a plain text and cannot be easily automated.

A CML-text

From chapter 1, § 2 of E. Landau's *Foundations of Analysis* Landau [1951].

Theorem 6. [Commutative Law of Addition]

$$x + y = y + x.$$

Proof Fix y, and $\mathfrak{M}$ be the set of all x for which the assertion holds.
I) We have

$$y + 1 = y',$$

and furthermore, by the construction in the proof of Theorem 4,

$$1 + y = y',$$

so that

$$1 + y = y + 1$$

and 1 belongs to $\mathfrak{M}$.
II) If x belongs to $\mathfrak{M}$, then

$$x + y = y + x,$$

Therefore

$$(x + y)' = (y + x)' = y + x'.$$

By the construction in the proof of Theorem 4, we have

$$x' + y = (x + y)',$$

hence

$$x' + y = y + x',$$

so that x' belongs to $\mathfrak{M}$. The assertion therefore holds for all x. □

LaTeX code

draft documents	✓
public documents	✓
computations and proofs	✗

```
\begin{theorem}[Commutative Law of Addition]\label{theorem:6}
  $$x+y=y+x.$$
\end{theorem}
\begin{proof}
  Fix $y$, and $\mathfrak{M}$ be the set of all $x$ for which the
  assertion holds.
  \begin{enumerate}
  \item We have $$y+1=y',$$
    and furthermore, by the construction in
    the proof of Theorem~\ref{theorem:4}, $$1+y=y',$$
    so that $$1+y=y+1$$
    and $1$ belongs to $\mathfrak{M}$.
  \item If $x$ belongs to $\mathfrak{M}$, then $$x+y=y+x,$$
    Therefore
    $$(x+y)'=(y+x)'=y+x'.$$
    By the construction in the proof of
    Theorem~\ref{theorem:4}, we have $$x'+y=(x+y)',$$
    hence
    $$x'+y=y+x',$$
    so that $x'$ belongs to $\mathfrak{M}$.
  \end{enumerate}
  The assertion therefore holds for all $x$.
\end{proof}
```

The problem with formal logic

- *Frege, Begriffsschrift*: *I found the inadequacy of language to be an obstacle; no matter how unwieldy the expressions I was ready to accept, I was less and less able, as the relations became more and more complex, to attain precision*

- In 1879, he wrote the *Begriffsschrift*, whose *first purpose is to provide us with the most reliable test of the validity of a chain of inferences*.

- He wrote the *Grundlagen* and *Grundgesetze der Arithmetik* where mathematics is seen as a branch of logic and arithmetic is described in *Begriffsschrift*.

- In 1902, Russell wrote a letter to Frege (Heijenoort [1967]) informing him of a *paradox* (see Kamareddine et al. [2002]).

- To avoid the paradox, Russell used *type theory* in the famous *Principia Mathematica* (Whitehead and Russell $[1910^1, 1927^2]$) where mathematics was founded on logic.

- Advances were also made in *set theory* Zermelo [1908], *category theory* (MacLane [1972]), etc., each being advocated as a better foundation for mathematics.

- But, none of the logical languages of the 20th century satisfies the criteria expected of a language of mathematics.
  - A logical language does not have *mathematico-linguistic* categories, is *not universal* to all mathematicians, and is *not a satisfactory communication medium*.
  - Logical languages make fixed choices (*first versus higher order, predicative versus impredicative, constructive versus classical, types or sets*, etc.). But different parts of mathematics need different choices and there is no universal agreement as to which is the best formalism.
  - A logician writes in logic their *understanding* of a mathematical-text as a formal, complete text which is structured considerably *unlike* the original, and is of little use to the *ordinary* mathematician.
  - Mathematicians do not want to use formal logic and have *for centuries* done mathematics without it.
- *So, mathematicians kept to* CML.
- We would like to find an alternative to CML which avoids some of the features of the logical languages which made them unattractive to mathematicians.

The problem with fully checked proofs (on computer)

- In 1967 the famous mathematician de Bruijn began work on logical languages for complete books of mathematics that can be *fully* checked by machine.
- People are prone to error, so if a machine can do proof checking, we expect fewer errors.
- Most mathematicians doubted de Bruijn could achieve success, and computer scientists had no interest at all.
- However, he persevered and built *Automath* (AUTOmated MATHematics).
- Today, there is much interest in many approaches to proof checking for verification of computer hardware and software.
- Many theorem provers have been built to mechanically check mathematics and computer science reasoning (e.g. Isabelle, HOL, Coq, etc.).

- A CML-text is structured differently from a computer-checked text proving the same facts. *Making the latter involves extensive knowledge and many choices:*
  - First, the needed choices include:
    - The choice of the *underlying logical system.*
    - The choice of *how concepts are implemented* (equational reasoning, equivalences and classes, partial functions, induction, etc.).
    - The choice of the *formal system*: a type theory (dependent?), a set theory (ZF? FM?), a category theory? etc.
    - The choice of the *proof checker*: Automath, Isabelle, Coq, PVS, Mizar...
  - Any informal reasoning in a CML-text will cause headaches as it is hard to turn a big step into a (series of) syntactic proof expressions.
  - Then the CML-text is *reformulated* in a fully *complete* syntactic formalism where every detail is spelled out. Very long expressions replace a clear CML-text. The new text is useless to ordinary mathematicians.

- So, *automation is user-unfriendly* for the mathematician/computer scientist.

- It is the hope that the alternative to CML may help in dividing the jump from informal mathematics to a fully formal one into smaller more informed steps.

Coq

draft documents	✗
public documents	✗
computations and proofs	✓

From Module `Arith.Plus` of Coq standard library (`http://coq.inria.fr/`).

```
Lemma plus_sym : (n,m:nat)(n+m)=(m+n).
Proof.
Intros n m ; Elim n ; Simpl_rew ; Auto with arith.
Intros y H ; Elim (plus_n_Sm m y) ; Simpl_rew ; Auto with arith.
Qed.
```

Where do we start? de Bruijn's Mathematical Vernacular MV

- *De Bruijn's Automath not just [...] as a technical system for verification of mathematical texts, it was rather a life style with its attitudes towards understanding, developing and teaching mathematics....The way mathematical material is to be presented to the system should correspond to the usual way we write mathematics. The only things to be added should be details that are usually omitted in standard mathematics.*

- MV is faithful to CML yet is formal and avoids ambiguities.

- MV is close to the usual way in which mathematicians write.

- MV has a syntax based on linguistic categories not on set/type theory.

- MV is weak as regards correctness: the rules of MV mostly concern *linguistic* correctness, its types are mostly linguistic so that the formal translation into MV is satisfactory *as a readable, well-organized text*.

Problems with MV

- MV makes many logical and mathematical choices which are best postponed.

- MV incorporates certain correctness requirements, there is for example a hierarchy of types corresponding with sets and subsets.

- MV is already *on its way* to a full formalization, while we want to remain *closer to* a given informal mathematical content.

- We want a $formal$ language *MathLang* which •has the advantages of CML but not its disadvantages and •respects CML content.

- The above items mean that MV fails in this aim.

What is the aim for MathLang?

Can we formalise a CML text avoiding as much as possible the ambiguities of natural language while still guaranteeing the following four goals?

1. The formalised text looks very much like the original CML text (and hence the content of the original CML text is respected).

2. The formalised text can be fully manipulated and searched.

3. Steps can be made to do computation (via computer algebra systems) and proof checking (via proof checkers) on the formalised text.

4. This formalisation of text is as simple a process to the mathematician as LaTeX is.

Starting point for MathLang: MV and WTT

- MV is the driving force behind MathLang. But MV fails on goal 1.
- Weak Type Theory, WTT Kamaredine and Nederpelt [2004], started from MV, but attempted to avoid its problems.
- WTT was intended as a *a 2nd language* for mathematicians which *satisfies* many criteria:

a1. WTT is *formal, suitable for computerization.*

a2. WTT helps mathematicians precisely identify the structure where they work.

a3. WTT makes an expert/teacher/student *aware of the complexity of a mathematical notion.*

a4. WTT encourages thinking about the interdependencies of notions (e.g., in which part of the chapter the definition holds), *so WTT texts are better structured than* CML *texts.*

a5. WTT respects all linguistic categories in the special ways they are used by mathematicians, e.g., *nouns*, *adjectives*, etc.

a6. WTT does *not restrict* the mathematician to set/type/category theory.

a7. Unlike set/type theory, WTT has basic notions needed for *text* such as *definition*, *theorem*, *step in a proof*, *section*, etc.

a8. The ambiguities in the CML-texts disappear in the translation to WTT. For example, the anaphoric obscurities in CML are resolved in WTT by the strict context management.

a9. Although the CML text and its initial translation into WTT are incomplete, WTT has *additional* levels *supporting more rigor.*

One can define *further translations into more logically complete versions.*

Improvements of MathLang over WTT

- Although WTT succeeds in many ways and is a considerable improvement on MV, it still fails on goal 1. *A WTT text is not close to its original* CML.

- MathLang starts from WTT, *extends its syntax and adds natural language as a top level.*

- A *MathLang text remains* close *to its original* CML.

- The CML-text is covered *exactly* in its formal version in the MathLang-text.

- We are using MathLang to translate two CML-books Landau [1951]; Heath [1956]

Syntax of type free lambda calculus

- $\mathcal{V} = \{x, y, z, \dots\}$ is an infinite set of *term variables*. We let $v, v_1, v_2, v', v'', \cdots$ range over $\mathcal{V}$
- $\mathcal{M} ::= \mathcal{V} \mid (\lambda\mathcal{V}.\mathcal{M}) \mid (\mathcal{M}\mathcal{M})$. We let $A, B, C \cdots$ range over $\mathcal{M}$.
- *Examples* $(\lambda x.x)$, $(\lambda x.(xx))$, $(\lambda x.(\lambda y.x))$, $(\lambda x.(\lambda y.(xy)))$, and $((\lambda x.x)(\lambda x.x))$.
- This simple language is surprisingly rich. Its richness comes from the freedom to create and apply functions, especially higher order functions to other functions (and even to themselves).

Meaning of Terms

- *Assume* a model $\mathcal{D}$ of the lambda calculus. Let ENV $= \{\sigma \mid \sigma : \mathcal{V} \mapsto \mathcal{D}\}$
- *Variables* The meaning of a variable is determined by what the variable is bound to in the *environment*.
- Expressions have variables and variables take values according to environment.
- Example, if $\mathcal{V} = \{x, y, z\}$ and if $\mathcal{D}$ contains all natural numbers, then one possible environment might be σ where $\sigma(x) = 1$, $\sigma(y) = 3$ and $\sigma(z) = 1$.
- *Function application* If A and B are λ-expressions, then so is (AB). This expression denotes the result of applying the function denoted by A to the meaning of B.
- For example, if A denotes the identity function and B denotes the number 3 then AB denotes identity applied to 3 which is 3.
- *Abstraction* $\lambda v.A$ denotes the function which takes an object a and returns the result of applying the function denoted by A in an environment in which V denotes a.

The semantic function

- Let $\sigma(a/v) : \mathcal{V} \mapsto \mathcal{D}$ where
  $\sigma(a/v)(v') = \sigma(v')$ if $v \neq v'$ and $\sigma(a/v)(v) = a$
- Let $[\,]\ : \mathcal{M} \mapsto$ ENV $\mapsto D$.
- $[v]_\sigma = \sigma(v)$.
- $[AB]_\sigma = [A]_\sigma([B]_\sigma)$.
- $[(\lambda v.A)]_\sigma = f$ where $f : \mathcal{D} \mapsto \mathcal{D}$ and $f(a) = [A]_{\sigma(a/v)}$.
- *Example:* $[(\lambda x.x)]_\sigma = f$ where $f(a) = [x]_{\sigma(a/x)} = \sigma(a/x)(x) = a$.
- Hence, $(\lambda x.x)$ denotes the identity function.

Exercises

- Exercise, show that $(\lambda x.(\lambda y.x))$ denotes the function which takes two arguments and returns the first.

- Represent the following mathematical functions in the λ-calculus:

  1. $f : x \rightarrow g$ where $g : y \rightarrow x + y$.
  2. $f : x \rightarrow x + y$ and $g : y \rightarrow x + y$.
  3. The function f that takes three functions g, h, k and composes them.
  4. The function f that takes a function g and iterates it five times.

- Describe the functions denoted by $(\lambda x.(\lambda y.(xy)))$, $(\lambda x.(\lambda y.y))$ and $(\lambda x.(\lambda y.x))$.

Notational Conventions

- Functional application associates to the left. So ABC denotes $((AB)C)$.
- The body of a λ is anything that comes after it. So, instead of $(\lambda v.(A_1 A_2 \ldots A_n))$, we write $\lambda v.A_1 A_2 \ldots A_n$.
- A sequence of λ's is compressed to one, so $\lambda xyz.t$ denotes $\lambda x.(\lambda y.(\lambda z.t))$.

As a consequence of these notational conventions we get:

- Parentheses may be dropped: (AB) and $(\lambda v.A)$ are written AB and $\lambda v.A$.
- Application has priority over abstraction: $\lambda x.yz$ means $\lambda x.(yz)$ and not $(\lambda x.y)z$.

Free and Bound Variables

- Evaluating $(\lambda fx.fx)g$ to $\lambda x.gx$ is perfectly acceptable but evaluating $(\lambda fx.fx)x$ to $\lambda x.xx$ is not.

- Check the meaning of these two expressions. $\lambda x.gx$ takes a and applies g to a. $\lambda x.xx$ takes a and applies it to itself.

- Also, $(\lambda fx.fx)$ is the same as $(\lambda fy.fy)$ but is it correct to evaluate $(\lambda fx.fx)x$ to $\lambda x.xx$ and $(\lambda fy.fy)x$ to $\lambda y.xy$? Shouldn't $(\lambda fx.fx)x$ be equal to $(\lambda fy.fy)x$?

- We define the notions of *free* and *bound* variables which will play an important role in avoiding the problem above. The free x in $(\lambda fx.fx)x$ should remain free in the result.

-

$$
\begin{array}{llll}
FV(v) & =_{def} & \{v\} & BV(v) & =_{def} & \emptyset \\
FV(\lambda v.A) & =_{def} & FV(A) - \{v\} & BV(\lambda v.A) & =_{def} & BV(A) \cup \{v\} \\
FV(AB) & =_{def} & FV(A) \cup FV(B) & BV(AB) & =_{def} & BV(A) \cup BV(B)
\end{array}
$$

- Exercise: In $(\lambda y.x(\lambda x.x))$ which variables are bound and which are free?

Substitution

- For any A, B, v, we define $A[v := B]$ to be the result of substituting B for every free occurrence of v in A, as follows:

$$
\begin{array}{lcl}
v[v := B] & \equiv & B \\
v'[v := B] & \equiv & v \quad \text{if } v \not\equiv v' \\
(AC)[v := B] & \equiv & A[v := B]C[v := B] \\
(\lambda v.A)[v := B] & \equiv & \lambda v.A \\
(\lambda v'.A)[v := B] & \equiv & \lambda v'.A[v := B] \\
 & & \text{if } v' \not\equiv v \text{ and } (v' \notin FV(B) \text{ or } v \notin FV(A)) \\
(\lambda v'.A)[v := B] & \equiv & \lambda v''.A[v' := v''][v := B] \\
 & & \text{if } v' \not\equiv v \text{ and } (v' \in FV(B) \text{ and } v \in FV(A))
\end{array}
$$

- So, in $(\lambda x.fx)[f := x]$, as $x \in FV(x)$ and $f \in FV(fx)$, we use last clause and get $(\lambda x.fx)[x := y][f := x] = (\lambda y.fy)[f := x] = (\lambda y.xy)$.

- Calculate $(\lambda x.y)[y := x]$. Why do we disallow the result to be $\lambda x.x$?

An easier alternative

- Use Barendregt's convention (BC) where in any term, free variables are called differently from bound variables.

- So, $(\lambda fx.fx)x$ must be written as $(\lambda fy.fy)x$ or $(\lambda fz.fz)x$, etc. and hence reducing $(\lambda fy.fy)x$ results correctly in $\lambda y.fy$.

- Similarly, $(\lambda x.y)[y := x]$ must be rewritten to $(\lambda z.y)[y := x]$

Exercises

- Evaluate:

  1. $(\lambda x.xy)[x := \lambda z.z]$
  2. $(\lambda y.x(\lambda x.x))[x := \lambda y.yx]$
  3. $(y(\lambda z.xz))[x := \lambda y.zy]$

- Check that:

  - $(\lambda y.yx)[x := z] \equiv \lambda y.yz$,
  - $(\lambda y.yx)[x := y] \equiv \lambda z.zy$,
  - $(\lambda y.yz)[x := \lambda z.z] \equiv \lambda y.yz$.

ALPHA Reduction

- *Compatibility*

$$\frac{A \rightarrow B}{AC \rightarrow BC} \qquad \frac{A \rightarrow B}{CA \rightarrow CB} \qquad \frac{A \rightarrow B}{\lambda v.A \rightarrow \lambda v.B}$$

- $\rightarrow_\alpha$ is defined to be the least compatible relation closed under the axiom:

$$(\alpha) \qquad \lambda v.A \rightarrow_\alpha \lambda v'.A[v := v'] \qquad \text{where } v' \notin FV(A)$$

- $\twoheadrightarrow_\alpha$ is the reflexive, transitive closure of $\rightarrow_\alpha$.

- $=_\alpha$ is the reflexive, transitive, symmetric closure of $\rightarrow_\alpha$.

- $\lambda x.x \rightarrow_\alpha \lambda y.y$ but it is not the case that $\lambda x.xy \rightarrow_\alpha \lambda y.yy$. Moreover, $\lambda z.(\lambda x.x)x \twoheadrightarrow_\alpha \lambda z.(\lambda y.y)x$.

- $\lambda x.x =_\alpha \lambda y.y$.

BETA Reduction

- $\rightarrow_\beta$ is defined to be the least compatible relation closed under the axiom:

$$(\beta) \qquad (\lambda v.A)B \rightarrow_\beta A[v := B]$$

- $\rightarrow_\beta$ is the reflexive transitive closure of $\rightarrow_\beta$.
- $=_\beta$ is the reflexive transitive, symmetric closure of $\rightarrow_\beta$.
- We say that A is in β-normal form if there is no B such that $A \rightarrow_\beta B$.
- Check that:
  - $(\lambda x.x)(\lambda z.z) \rightarrow_\beta \lambda z.z$,
  - $(\lambda y.(\lambda x.x)(\lambda z.z))xy \rightarrow_\beta y$,
  - both $\lambda z.z$ and y are β-normal forms.

Exercises

- Give the sets of free and bound variables of the following λ-terms and for each variable occurrence, say whether it is bound or free:

  1. $\lambda x.\lambda y.(\lambda y.(\lambda z.z)\lambda y.z)y$
  2. $(\lambda x.(\lambda y.\lambda z.pq)y)xz$
  3. $\lambda x.yz(\lambda yz.y)x$

  For each of the above terms apply β-reduction until no β-redexes can be found.

Metatheory

- *Some Programs loop/don't terminate:* $\underline{(\lambda x.xx)(\lambda x.xx)}$ does not have a normal form.

- *We can evaluate programs in different orders, but always get the same final result:*
$(\lambda y.\underline{(\lambda x.x)(\lambda z.z)})xy \rightarrow_\beta \underline{(\lambda y.\lambda z.z)xy} \rightarrow_\beta \underline{(\lambda z.z)y} \rightarrow_\beta y$ and
$\underline{(\lambda y.(\lambda x.x)(\lambda z.z))xy} \rightarrow_\beta (\underline{(\lambda x.x)(\lambda z.z)})y \rightarrow_\beta \underline{(\lambda z.z)y} \rightarrow_\beta y$

- *The order we use to evaluate programs can affect termination:*
A term may be normalising but not strongly normalising:
$\underline{(\lambda y.z)((\lambda x.xx)(\lambda x.xx))} \rightarrow_\beta z$ yet
$(\lambda y.z)(\underline{(\lambda x.xx)(\lambda x.xx)}) \rightarrow_\beta (\lambda y.z)(\underline{(\lambda x.xx)(\lambda x.xx)}) \rightarrow_\beta \ldots$

- A *program may grow after reduction:*

$$\begin{aligned}
\underline{(\lambda x.xxx)(\lambda x.xxx)} \quad &\rightarrow_\beta \quad \underline{(\lambda x.xxx)(\lambda x.xxx)}(\lambda x.xxx) \\
&\rightarrow_\beta \quad \underline{(\lambda x.xxx)(\lambda x.xxx)}(\lambda x.xxx)(\lambda x.xxx) \\
&\rightarrow_\beta \quad \ldots
\end{aligned}$$

- If an expression β-reduces in two different ways to two values, then those values, if they are in β-normal form are the same (up to α-conversion).

- $\underline{(\lambda xyz.xz(yz))(\lambda x.x)}(\lambda x.x) \rightarrow_\beta (\lambda yz.\underline{(\lambda x.x)z}(yz))(\lambda x.x) \rightarrow_\beta$ $\underline{(\lambda yz.z(yz))(\lambda x.x)} \rightarrow_\beta \lambda z.z(\underline{(\lambda x.x)z}) \rightarrow_\beta \lambda z.zz.$

- $\underline{(\lambda xyz.xz(yz))(\lambda x.x)}(\lambda x.x) \rightarrow_\beta \underline{(\lambda yz.(\lambda x.x)z(yz))(\lambda x.x)} \rightarrow_\beta$ $\lambda z.\underline{(\lambda x.x)z}((\lambda x.x)z) \rightarrow_\beta \lambda z.z(\underline{(\lambda x.x)z}) \rightarrow_\beta \lambda z.zz.$

- $\underline{(\lambda xyz.xz(yz))(\lambda x.x)}(\lambda x.x) \rightarrow_\beta \underline{(\lambda yz.(\lambda x.x)z(yz))(\lambda x.x)} \rightarrow_\beta$ $\lambda z.(\lambda x.x)z(\underline{(\lambda x.x)z}) \rightarrow_\beta \lambda z.\underline{(\lambda x.x)z}z \rightarrow_\beta \lambda z.zz.$

- *Church-Rosser Theorem*
  $\forall A, B, C \in \mathcal{M}\ \exists D \in \mathcal{M}\ : (A \twoheadrightarrow_\beta B \wedge A \twoheadrightarrow_\beta C) \Rightarrow (B \twoheadrightarrow_\beta D \wedge C \twoheadrightarrow_\beta D)\,.$

Call by Name and Call by Value

- *Normal Order/Call by name:* At every stage, reduce the leftmost-outermost redex. E.g., $\underline{(\lambda y.y)((\lambda x.x)1)} \to \underline{(\lambda x.x)1} \to 1$.

- *Applicative Order/Call by value:* At every stage, reduce the leftmost-innermost redex. E.g., $(\lambda y.y)(\underline{(\lambda x.x)1}) \to \underline{(\lambda y.y)1} \to 1$.

- *If a program terminates, call by name reduction will reach final value but call by value may not. Call by value is faster than call by name.*

- *Call by Name:* $\underline{(\lambda y.z)((\lambda x.xx)(\lambda x.xx))} \to_\beta z$ yet

- *Call by Value:* $(\lambda y.z)(\underline{(\lambda x.xx)(\lambda x.xx)}) \to_\beta (\lambda y.z)(\underline{(\lambda x.xx)(\lambda x.xx)}) \to_\beta \ldots$

- *Call by Value:* $(\lambda x.xx)(\underline{(\lambda y.y)(\lambda z.z)}) \to \underline{(\lambda x.xx)(\lambda z.z)} \to \underline{(\lambda z.z)(\lambda z.z)} \to (\lambda z.z)$.

- *Call by Name:* $\underline{(\lambda x.xx)((\lambda y.y)(\lambda z.z))} \to (\underline{(\lambda y.y)(\lambda z.z)})((\lambda y.y)(\lambda z.z)) \to \underline{(\lambda z.z)((\lambda y.y)(\lambda z.z))} \to \underline{(\lambda y.y)(\lambda z.z)} \to (\lambda z.z)$

Booleans in λ-calculus

$$
\begin{array}{lcl}
\text{true} & \equiv & \lambda xy.x \\
\text{false} & \equiv & \lambda xy.y \\
\text{not} & \equiv & \lambda x.x \text{ false } \text{ true} \\
\text{and} & \equiv & \lambda xy.xy \text{ false} \\
\text{or} & \equiv & \lambda xy.x \text{ true } y \\
\text{if } M \text{ then } N_1 \text{ else } N_2 & \equiv & MN_1N_2
\end{array}
$$

$$
\begin{array}{lcl}
\text{and true true} & =_\beta & \text{true true false} \\
& =_\beta & (\lambda xy.x) \text{ true false} \\
& =_\beta & (\lambda y. \text{ true}) \text{ false} \\
& =_\beta & \text{true}
\end{array}
$$

$$
\begin{array}{lcll}
\text{if true then } N_1 \text{ else } N_2 & =_\beta & \text{true } N_1N_2 & \\
& =_\beta & (\lambda y.N_1)N_2 & \\
& =_\beta & N_1 & \qquad \text{Note that } y \notin FV(N_1)
\end{array}
$$

Numerals in λ-calculus

- $0 \equiv \lambda xy.y$, $1 \equiv \lambda xy.xy$, $2 \equiv \lambda xy.x(xy)$ and so on.

$$\begin{array}{lcl} S & \equiv & \lambda xyz.xy(yz) \\ A & \equiv & \lambda xyzp.xz(yzp) \\ M & \equiv & \lambda xyz.x(yz) \\ E & \equiv & \lambda xy.yx \\ Z & \equiv & \lambda x.x(\text{true false})\text{true} \end{array}$$

$$\begin{array}{lcl} Sn & =_\beta & n+1 \\ Amn & =_\beta & m+n \\ Z0 & =_\beta & \text{true} \\ Z(Sn) & =_\beta & \text{false} \end{array}$$

Recursion in λ-calculus

- a is a fixed point of E if $Ea = a$.

- *Every program E (term of λ-calculus) has a fixed point:*

- Let $Y = (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))$ and let $a = (YE)$.

- *$Ea = a$:* because: $a = (YE) = \underline{(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))E} = \underline{(\lambda x.E(xx))(\lambda x.E(xx))} = E((\lambda x.E(xx))(\lambda x.E(xx))) = E(YE) = Ea$.

- With the presence of fixed points, we can solve recursive equations;

- fact $\equiv \lambda x.$ if Zx then 1 else $Mx($ fact $(Px))$

- fact is defined in terms of itself.

- Let $E \equiv \lambda yx.$ if Zx then 1 else $Mx(y(Px))$.
  As we see, E is defined in terms of things that already exist and not in terms of itself.

- Now, we take fact $\equiv (YE)$, and so, as $E(YE) = (YE)$ we have:
  fact $= E($fact$) = \lambda x.$ if Zx then 1 else $Mx($ fact $(Px))$.

PAIRING in λ-calculus

$$\begin{array}{lcl} \text{pair} & \equiv & \lambda xyz.zxy \\ \text{fst} & \equiv & \lambda x.x \text{ true} \\ \text{snd} & \equiv & \lambda x.x \text{ false} \end{array}$$

It is easy to prove that

- fst(pair $AB) = A$
- snd(pair $AB) = B$
- fst(pair $AB) = (\lambda x.x$ true)(pair $AB) =$ (pair AB) true $=$ pair AB true $=$ $(\lambda xyz.zxy)AB$ true $=$ true $AB = (\lambda xy.x)AB = A$.
- snd(pair $AB) = (\lambda x.x$ false)(pair $AB) =$ (pair AB) false $=$ pair AB false $=$ $(\lambda xyz.zxy)AB$ false $=$ false $AB = (\lambda xy.y)AB = B$.

LISTS in λ-calculus

- The equation $xy = x$ has a solution $\perp$ where $\perp y = \perp$ for any y.

- Le $E = \lambda xy.x$ and let $\perp = YE$. Then $YE = E(YE)$.
  So, $\perp = E\perp$ and $\perp y = E\perp y = (\lambda xy.x)\perp y = \perp$.

$$
\begin{array}{lcl}
\text{null} & \equiv & \text{fst} \\
[\,] & \equiv & \text{pair}\ \ \text{true}\ \perp \\
[E] & \equiv & \text{pair}\ \ \text{false}\ (\text{pair}\ E\ [\,]) \\
[E_1, \ldots, E_n] & \equiv & \text{pair}\ \ \text{false}\ (\text{pair}\ E_1\ [E_2, \ldots E_n])
\end{array}
$$

- *Exercise:* null $[\,] =$ true and null $[E_1, \ldots, E_n] =$ false

- null $[\,] =$ fst(pair true $\perp$) $=$ true .

- null $[E_1, \ldots, E_n] =$ fst(pair false (pair E_1 $[E_2, \ldots E_n]$)) $=$ false

$$
\begin{array}{lcl}
\text{hd} & \equiv & \lambda l.\ \text{if (null } l) \text{ then } \bot \text{ else (fst (snd } l)) \\
\text{tl} & \equiv & \lambda l.\ \text{if (null } l) \text{ then } \bot \text{ else (snd (snd } l)) \\
\text{cons} & \equiv & \lambda x l.\ \text{pair false (pair } x\ l)
\end{array}
$$

Exercises:

- null (cons xl) $=$ fst $((\lambda xl.$ pair false (pair $xl))xl) =$ fst (pair false (pair $xl)) =$ false

- snd (cons $x\ l$) $=$ snd (pair false (pair $x\ l$)) $=$ (pair $x\ l$)

- hd (cons $x\ l$) $= (\lambda l.$ if (null l) then $\bot$ else (fst (snd l)))(cons $x\ l$) $=$
  if (null (cons $x\ l$)) then $\bot$ else (fst (snd (cons $x\ l$))) $=$
  if false then $\bot$ else (fst (snd (cons $x\ l$))) $=$
  fst (snd (cons $x\ l$)) $=$ fst (pair $x\ l$) $= x$

- tl (cons $x\,l$) $=$ ($\lambda l.$ if (null l) then $\perp$ else (snd (snd l)))(cons $x\,l$) $=$
  if (null (cons $x\,l$)) then $\perp$ else (snd (snd (cons $x\,l$))) $=$
  if false then $\perp$ else (snd (snd (cons $x\,l$))) $=$
  snd (snd (cons $x\,l$)) $=$ snd (pair $x\,l$) $= l$

- Find a λ-expression *append* such that

$$\text{append } x\, y = \text{ if (null } x) \text{ then } y \text{ else (cons(hd } x)(\text{append (tl } x)y)$$

- Let $E = \lambda axy.$ if (null x) then y else (cons (hd x)(a(tl x)y) and let append $= YE$.

- Then, append $xy = YExy = E(YE)xy = E(\text{append})xy =$ ($\lambda axy.$ if (null x) then y else (cons (hd x)(a(tl x)y)))append $xy =$ if (null x) then y else (cons (hd x)(append(tl x)y)

UNDECIDABILITY of HALTING

- *Note that $\perp$ loops*: $\perp = YE = E(YE) = E(E(YE)) = E(E(E(YE)))\ldots.$
- Let halts $E =$ true if E has a normal form and halts $E =$ false otherwise.
- halts is *Not* definable in the λ-calculus.
- Assume the contrary (i.e. halts is a λ-term), then
- Let foo $= \lambda x.$ if (halts x) then $\perp$ else 0.
- Let W be a solution to $x =$ foo x. Hence, $W =$ foo $W =$ if (halts W) then $\perp$ else 0.
- *Case halts W is true* then $W =$ if (halts W) then $\perp$ else $0 = \perp$.
  Absurd as $\perp$ does not have normal form.

- *Case halts W is false* then $W = \text{if } (\text{halts } W) \text{ then } \perp \text{ else } 0 = 0$.
  Absurd as 0 does have a normal form.

- Hence, what we assumed is false and so, halts is not a lambda term.

“propositions as types” or “proofs as terms”

- In this method proofs are first-class citizens of the logical system, whilst for many other logical systems, proofs are rather complex objects outside the logic (for example: derivation trees), and therefore cannot be easily manipulated.

- Heyting [1934a] describes the proof of an implication $a \Rightarrow b$ as: Deriving a solution for the problem b from the problem a.

- Kolmogorov [1932a] is even more explicit, and describes a proof of $a \Rightarrow b$ as the construction of a method that transforms each proof of a into a proof of b.

- This means that a proof of $a \Rightarrow b$ can be seen as a *(constructive) function* from the proofs of a to the proofs of b.

- In other words, the proofs of the proposition $a \Rightarrow b$ form exactly the set of functions from the set of proofs of a to the set of proofs of b.

- This suggests to identify a proposition with the set of its proofs.

  Now *types* are used to represent these sets of proofs. An element of such a set of proofs is represented as a *term* of the corresponding type.

  This means that propositions are interpreted as *types*, and proofs of a proposition a as *terms of type* a.

- $\textsc{Pat}$ was, independently from Heyting and Kolmogorov, discovered by Curry and Feys [1958a]

- Curry describes the so-called F-objects, which correspond more or less to the simple types of Church [1940b].

PAT with Howard

Howard [1980a] follows Curry and Feys [1958a] and combines it with Tait's correspondence between cut elimination and β-reduction of λ-terms Tait [1965].

Example 1. The following derivation of a proposition B:

$$\dfrac{\dfrac{\begin{array}{c}[A]\\ \boxed{\mathfrak{D}_1}\\ B\end{array}}{A \to B} \qquad \begin{array}{c}\boxed{\mathfrak{D}_2}\\ A\end{array}}{B}$$

can be transformed into:

$$\begin{array}{c}\boxed{\mathfrak{D}_2}\\ A\\ \boxed{\mathfrak{D}_1}\\ B\end{array}$$

We can decorate the two derivations above with λ-terms that represent proofs. This results in the following two deductions:

$$\dfrac{\dfrac{\begin{array}{c}[x{:}A]\\ \boxed{\mathfrak{D}_1}\\ T : B\end{array}}{(\lambda x{:}A.T) : (A \to B)} \qquad \begin{array}{c}\boxed{\mathfrak{D}_2}\\ S : A\end{array}}{((\lambda x{:}A.T)S) : B}$$

$$\begin{array}{c}\boxed{\mathfrak{D}_2}\\ S : A\\ \boxed{\mathfrak{D}_1}\\ T[x{:=}S] : B\end{array}$$

We see that the proof transformation exactly corresponds to the β-reduction

$$(\lambda x{:}A.T)S \to_\beta T[x{:=}s]$$

Church's Simply Typed λ-calculus $\lambda\!\to$ 1940

- *Types* • *Basic* individuals/propositions • *Arrows* $\alpha \to \beta$

  Examples of types: $(\alpha \to \beta) \to (\beta \to \gamma)$, $\alpha \to (\beta \to \gamma)$, $Bool \to Bool$.

- *Terms* *variables*, AB, $\lambda x{:}\alpha.A$.

- *(β)* $(\lambda x{:}\alpha.A)B \to_\beta A[x := B]$.

- *Start* If $(x : \alpha) \in \Gamma$ then $\Gamma \vdash x : \alpha$.

- *$\to$-introduction* If $\Gamma, x{:}\alpha \vdash A : \beta$ then $\Gamma \vdash \lambda x{:}\alpha.A : \alpha \to \beta$

- *$\to$-elimintation* If $\Gamma \vdash A : \alpha \to \beta$ and $\Gamma \vdash B : \alpha$ then $\Gamma \vdash AB : \beta$

- $\lambda x : \alpha.x : \alpha \to \alpha$.

  $\lambda x : (\alpha \to \beta).x : (\alpha \to \beta) \to (\alpha \to \beta)$.

  $\lambda x : \alpha.\lambda y : \beta.x : \alpha \to (\beta \to \alpha)$.

- $$\begin{array}{rcll} x:\alpha & \vdash & x:\alpha & \text{start} \\ & \vdash & \lambda x:\alpha.x:\alpha\to\alpha & \to\text{-introduction} \end{array}$$

- $$\begin{array}{rcll} x:\alpha\to\beta & \vdash & x:\alpha\to\beta & \text{start} \\ & \vdash & \lambda x:\alpha\to\beta.x:(\alpha\to\beta)\to(\alpha\to\beta) & \to\text{-introduction} \end{array}$$

- $$\begin{array}{lcll} x:\alpha,y:\beta & \vdash & x:\alpha & \text{start} \\ x:\alpha & \vdash & \lambda y:\beta.x:\beta\to\alpha & \to\text{-introduction} \\ & \vdash & \lambda x:\alpha.(\lambda y:\beta.x):\alpha\to(\beta\to\alpha) & \to\text{-introduction} \end{array}$$

- But, $\lambda x :?.xx$ cannot be typed.

- In $\lambda\!\rightarrow$, the function which takes $f : \mathbb{N} \rightarrow \mathbb{N}$ and $x : \mathbb{N}$ and returns $f(f(x))$ is:

$$\lambda f : \mathbb{N} \rightarrow \mathbb{N}.\lambda x : \mathbb{N}.f(f(x))$$

  and has type

$$(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$$

- If we want the same function on booleans, we would need to write:

$$\lambda f : \mathcal{B} \rightarrow \mathcal{B}.\lambda x : \mathcal{B}.f(f(x))$$

  which has type

$$(\mathcal{B} \rightarrow \mathcal{B}) \rightarrow (\mathcal{B} \rightarrow \mathcal{B})$$

- Instead of repeating the work, we can write the *Polymorphic* doubling function as:

$$\lambda \alpha : *.\lambda f : \alpha \rightarrow \alpha.\lambda x : \alpha.f(f(x))$$

- Now, we can instantiate α to what we need:

- $\alpha = \mathbb{N}$ then:
  $(\lambda\alpha : *.\lambda f : \alpha \to \alpha.\lambda x : \alpha.f(f(x)))\mathbb{N} = \lambda f : \mathbb{N} \to \mathbb{N}.\lambda x : \mathbb{N}.f(f(x))$.

- $\alpha = \mathcal{B}$ then:
  $(\lambda\alpha : *.\lambda f : \alpha \to \alpha.\lambda x : \alpha.f(f(x)))\mathcal{B} = \lambda f : \mathcal{B} \to \mathcal{B}.\lambda x : \mathcal{B}.f(f(x))$.

- $\alpha = (\mathcal{B} \to \mathcal{B})$ then: $(\lambda\alpha : *.\lambda f : \alpha \to \alpha.\lambda x : \alpha.f(f(x)))(\mathcal{B} \to \mathcal{B}) =$
  $\lambda f : (\mathcal{B} \to \mathcal{B}) \to (\mathcal{B} \to \mathcal{B}).\lambda x : (\mathcal{B} \to \mathcal{B}).f(f(x))$.

- So, types can be abstracted over (like for terms) and we can pass types as arguments (like for terms).

- But, as we have new terms like $\lambda\alpha : *.\lambda f : \alpha \to \alpha.\lambda x : \alpha.f(f(x))$, we need to say what their types is.

- The type of this function is: $\Pi\alpha : *.(\alpha \to \alpha) \to (\alpha \to \alpha)$.

Common features of modern types and functions

- We can *construct* a type by abstraction. (Write $\alpha : *$ for α *is a type*)
  - $\lambda y : \alpha.y$, the identity over α *has type* $\alpha \to \alpha$
  - $\lambda\alpha : *.\lambda y : \alpha.y$, the polymorphic identity *has type* $\Pi\alpha : *.\alpha \to \alpha$

- We can *instantiate* types. E.g., if $\alpha = \mathbb{N}$, then the identity over $\mathbb{N}$
  - $(\lambda y : \alpha.y)[\alpha := \mathbb{N}]$ *has type* $(\alpha \to \alpha)[\alpha := \mathbb{N}]$ or $\mathbb{N} \to \mathbb{N}$.
  - $(\lambda\alpha : *.\lambda y : \alpha.y)\mathbb{N}$ *has type* $(\Pi\alpha : *.\alpha \to \alpha)\mathbb{N} = (\alpha \to \alpha)[\alpha := \mathbb{N}]$ or $\mathbb{N} \to \mathbb{N}$.

- $(\lambda x{:}\alpha.A)B \to_\beta A[x := B]$ $\qquad$ $(\Pi x{:}\alpha.A)B \to_\Pi A[x := B]$

- Write $\alpha \to \alpha$ as $\Pi y : \alpha.\alpha$ when y not free in α.

Are we getting into self-application/Trouble?

- ML treats let val id $=$ (fn $x \Rightarrow x$) in (id id) end as this polymorphic term
  $(\lambda \mathsf{id}{:}(\Pi\alpha{:}{*}.\ \alpha \to \alpha).\ \mathsf{id}(\beta \to \beta)(\mathsf{id}\,\beta))(\lambda\alpha{:}{*}.\ \lambda x{:}\alpha.\ x)$

- The polymorphic identity function can be applied to its type too:
  $(\lambda\alpha : *.\lambda y : \alpha.y)(\Pi\alpha : *.\alpha \to \alpha) \to_\beta \lambda y : (\Pi\alpha : *.\alpha \to \alpha).y$

- So, we can now apply this result to polymorphic identity:
  $(\lambda y : (\Pi\alpha : *.\alpha \to \alpha).y)(\lambda\alpha : *.\lambda y : \alpha.y) \to_\beta (\lambda\alpha : *.\lambda y : \alpha.y)$

- *Problem??*

$$(\lambda\alpha : *.\lambda y : \alpha.y)(\Pi\alpha : *.\alpha \to \alpha)(\lambda\alpha : *.\lambda y : \alpha.y) \to_\beta (\lambda\alpha : *.\lambda y : \alpha.y)$$

- *THE NEW SYSTEM IS VERY SAFE*.

  Subject Reduction: If $\Gamma \vdash A : \alpha$ and $A \to_\beta A'$ then $\Gamma \vdash A' : \alpha$.

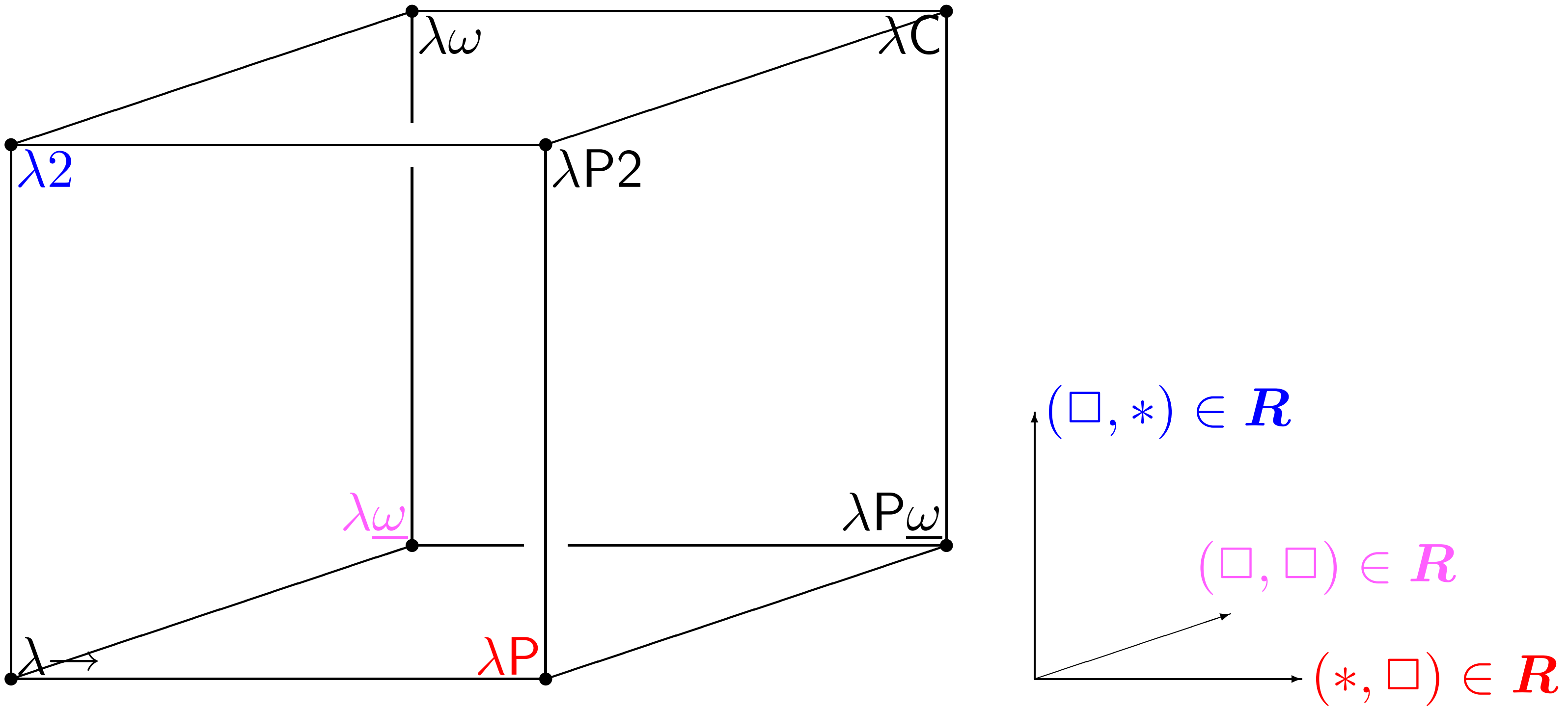
  Termination: If $\Gamma \vdash A : \alpha$ then both A and α terminate.

The Barendregt Cube

- Syntax: $A ::= v \mid * \mid \Box \mid AB \mid \lambda v{:}A.B \mid \Pi v{:}A.B$
- Formation rule: $$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_2}$$ if $(s_1, s_2) \in \boldsymbol{R}$

	Simple	Poly-morphic	Depend-ent	Constr-uctors	Related system
$\lambda\!\rightarrow$	$(*,*)$				λ^τ
$\lambda 2$	$(*,*)$	$(\Box,*)$			F
λP	$(*,*)$		$(*,\Box)$		AUT-QE, LF
$\lambda\underline{\omega}$	$(*,*)$			$(\Box,\Box)$	POLYREC
λP2	$(*,*)$	$(\Box,*)$	$(*,\Box)$		
$\lambda\omega$	$(*,*)$	$(\Box,*)$		$(\Box,\Box)$	Fω
λP$\underline{\omega}$	$(*,*)$		$(*,\Box)$	$(\Box,\Box)$	
λC	$(*,*)$	$(\Box,*)$	$(*,\Box)$	$(\Box,\Box)$	CC

The Barendregt Cube



Typing Polymorphic identity needs $(\square, *)$

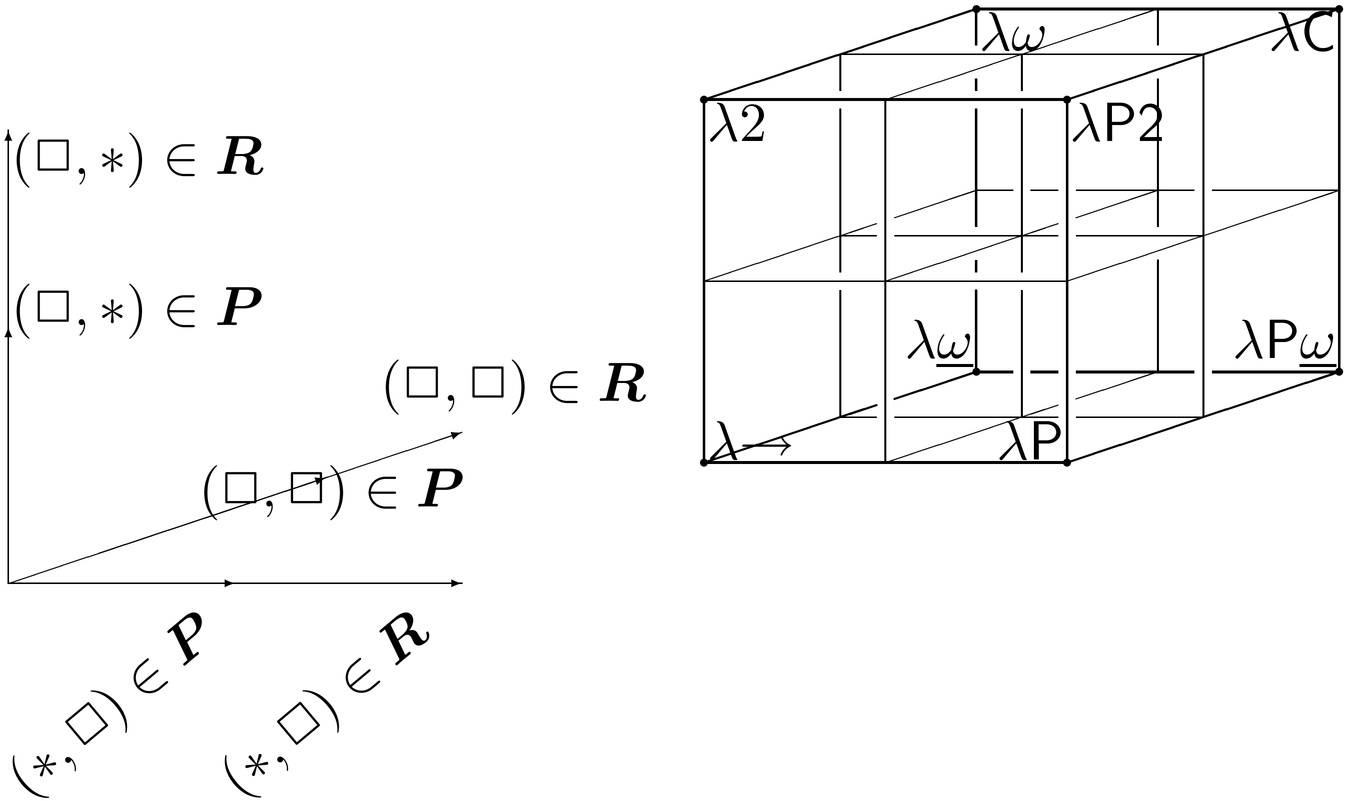
- $$\frac{y : * \vdash y : * \qquad y : *, x{:}y \vdash y : *}{y : * \vdash \Pi x{:}y.y : *}$$ by (Π) $(*, *)$

- $$\frac{y : *, x : y \vdash x : y \qquad y : * \vdash \Pi x{:}y.y : *}{y : * \vdash \lambda x : y.x : \Pi x{:}y.y}$$ by (λ)

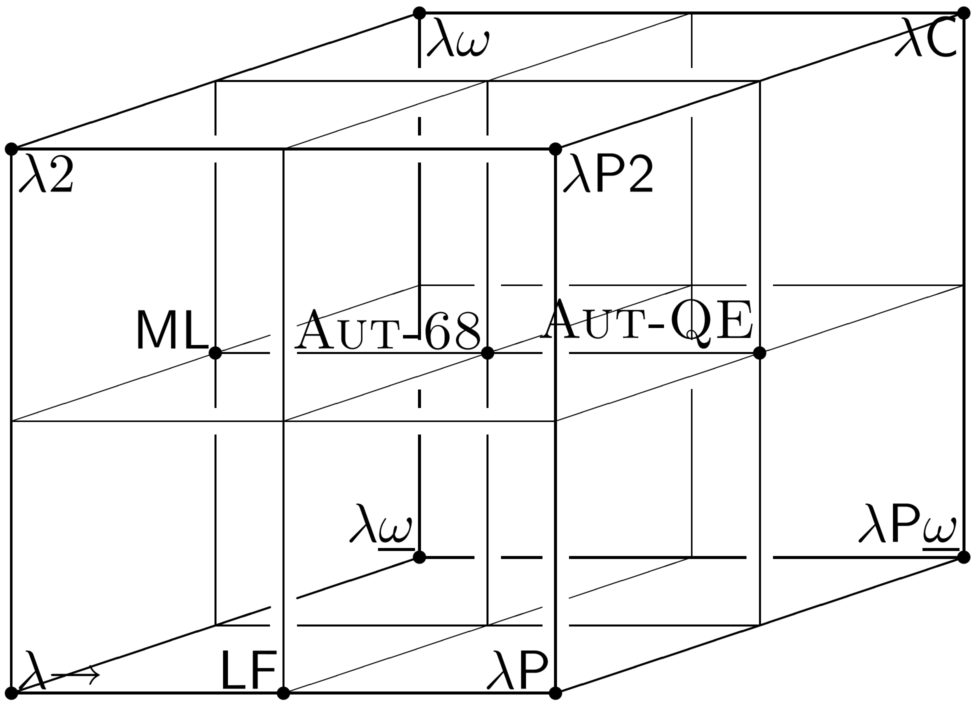
- $$\frac{\vdash * : \square \qquad y : * \vdash \Pi x{:}y.y : *}{\vdash \Pi y : *.\Pi x{:}y.y : *}$$ by (Π) $(\square, *)$

- $$\frac{y : * \vdash \lambda x : y.x : \Pi x{:}y.y \qquad \vdash \Pi y : *.\Pi x{:}y.y : *}{\vdash \lambda y : *.\lambda x : y.x : \Pi y : *.\Pi x{:}y.y}$$ by (λ)

The refined Barendregt Cube



ML in the refined Cube



MathLang

draft documents	✓
public documents	✓
computations and proofs	✓

- MathLang describes the grammatical and reasoning structure of mathematical texts

- A *weak type system* checks MathLang documents at a grammatical level

- MathLang eventually should support *all encoding uses*

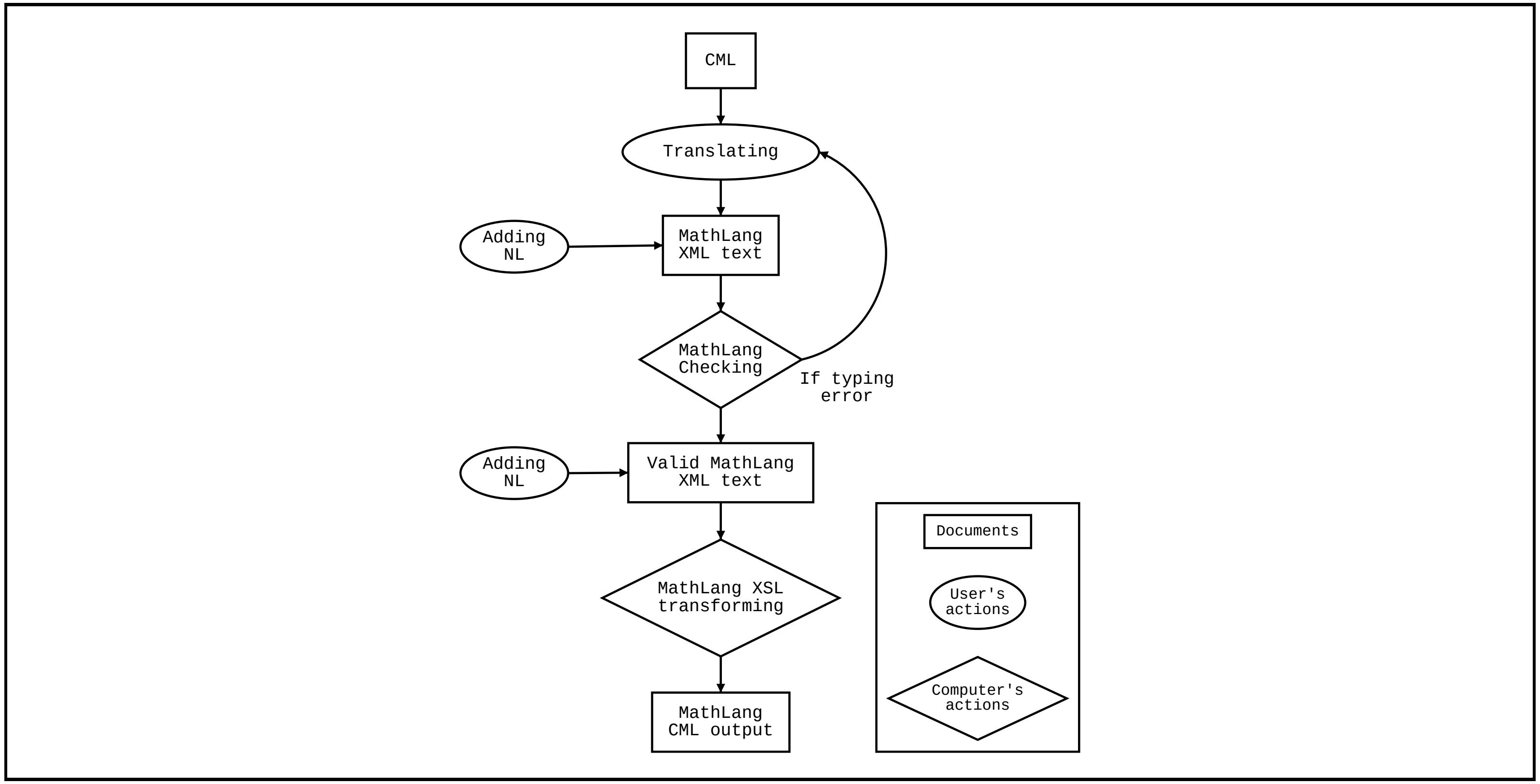


Figure 1: Translation

Weak Type Theory

In Weak Type Theory (or $\textsc{Wtt}$) we have the following linguistic categories:

- On the *atomic* level: *variables*, *constants* and *binders*,
- On the *phrase*[1] level: *terms* $\mathcal{T}$, *sets* $\mathbb{S}$, *nouns* $\mathcal{N}$ and *adjectives* $\mathcal{A}$,
- On the *sentence* level: *statements* P and *definitions* $\mathcal{D}$,
- On the *discourse* level: *contexts* $\mathbb{I}\Gamma$, *lines* $\mathbf{l}$ and *books* $\mathbf{B}$.

There is a hierarchy between these levels: atoms are part of phrases; atoms and phrases are part of sentences; and discourses are built from sentences.

[1]According to the Concise Oxford Dictionary, a phrase is *a group of words forming a conceptual unit, but not a sentence*, a discourse is *a connected series of utterances*.

Abstract Syntax of WTT

We use abstract syntax for the description of the various syntactic categories.

Example: $\mathbf{B} \;=\; \emptyset \mid \mathbf{B} \circ \mathbf{l}$ expresses that a book is either *the empty book* or a book $\mathbf{B}$ followed by a line $\mathbf{l}$. By convention, $\emptyset \circ \mathbf{l}$ is written as $\mathbf{l}$.

Binders are in the abstract form: $\mathtt{B}_{\mathcal{Z}}(\mathcal{E})$, where the $subscript\ \mathcal{Z}$ is a $declaration$ introducing a (bound) variable and its type, e.g. $x \in \mathbb{N}$.

- $\sum_{x\in\{0,1,\ldots,10\}}(x^2)$ and $\forall_{x\in\mathbb{N}}(x \geq 0)$ are examples of formulas with binders.

- The binding symbol for set comprehension, $\{\ldots \mid \ldots\}$, fits in this format after a slight modification. E.g., write $\{x \in \mathbb{R} | x > 5\}$ as $\mathtt{Set}_{x\in\mathbb{R}}(x > 5)$. *For uniformity, our standard for set notation will be the latter one.*

level	Main category	abstract syntax	Meta-symbol
atomic	*variables*	$\mathtt{V} = \mathtt{V}^T \vert \mathtt{V}^S \vert \mathtt{V}^P$	x
	constants	$\mathtt{C} = \mathtt{C}^T \vert \mathtt{C}^S \vert \mathtt{C}^N \vert \mathtt{C}^A \vert \mathtt{C}^P$	c
	binders	$\mathtt{B} = \mathtt{B}^T \vert \mathtt{B}^S \vert \mathtt{B}^N \vert \mathtt{B}^A \vert \mathtt{B}^P$	b
phrase	*terms*	$\mathcal{T} = \mathtt{C}^T(\overrightarrow{\mathcal{P}}) \vert \mathtt{B}^T_{\mathcal{Z}}(\mathcal{E}) \vert \mathtt{V}^T$	t
	sets	$\mathbb{S} = \mathtt{C}^S(\overrightarrow{\mathcal{P}}) \vert \mathtt{B}^S_{\mathcal{Z}}(\mathcal{E}) \vert \mathtt{V}^S$	s
	nouns	$\mathcal{N} = \mathtt{C}^N(\overrightarrow{\mathcal{P}}) \vert \mathtt{B}^N_{\mathcal{Z}}(\mathcal{E}) \vert \mathcal{A}\mathcal{N}$	n
	adjectives	$\mathcal{A} = \mathtt{C}^A(\overrightarrow{\mathcal{P}}) \vert \mathtt{B}^A_{\mathcal{Z}}(\mathcal{E})$	a
sentence	*statements*	$P = \mathtt{C}^P(\overrightarrow{\mathcal{P}}) \vert \mathtt{B}^P_{\mathcal{Z}}(\mathcal{E}) \vert \mathtt{V}^P$	S
	definitions	$\mathcal{D} = \mathcal{D}^\varphi \vert \mathcal{D}^P$	D
		$\mathcal{D}^\varphi = \mathtt{C}^T(\overrightarrow{V}) := \mathcal{T} \vert \mathtt{C}^S(\overrightarrow{V}) := \mathbb{S} \vert$ $\mathtt{C}^N(\overrightarrow{V}) := \mathcal{N} \vert \mathtt{C}^A(\overrightarrow{V}) := \mathcal{A}$	
		$\mathcal{D}^P = \mathtt{C}^P(\overrightarrow{V}) := P$	
discourse	*contexts*	$\mathbb{\Gamma} = \emptyset \mid \mathbb{\Gamma}, \mathcal{Z} \mid \mathbb{\Gamma}, P$	Γ
	lines	$\mathbf{l} = \mathbb{\Gamma} \triangleright P \mid \mathbb{\Gamma} \triangleright \mathcal{D}$	l
	books	$\mathbf{B} = \emptyset \mid \mathbf{B} \circ \mathbf{l}$	B

Figure 2: Main categories of syntax of WTT

Other category	abstract syntax	Meta-symbol
expressions	$\mathcal{E} = T \vert \mathbb{S} \vert \mathcal{N} \vert P$	E
parameters	$\mathcal{P} = T \vert \mathbb{S} \vert P$ (note: $\overrightarrow{\mathcal{P}}$ is a list of $\mathcal{P}$s)	P
typings	$\mathbf{T} = \mathbb{S} : \textsf{SET} \vert \mathcal{S} : \textsf{STAT} \vert T : \mathbb{S} \vert T : \mathcal{N} \vert T : \mathcal{A}$	T
declarations	$\mathcal{Z} = \mathtt{V}^S : \textsf{SET} \vert \mathtt{V}^P : \textsf{STAT} \vert \mathtt{V}^T : \mathbb{S} \vert \mathtt{V}^T : \mathcal{N}$	Z

Figure 3: Categories of syntax of WTT

Constants of WTT

The set $\mathtt{C} = \mathtt{C}^T|\mathtt{C}^S|\mathtt{C}^N|\mathtt{C}^A|\mathtt{C}^P$ is fixed, infinite and is disjoint from the set of variables. $\mathtt{C}$ is divided into the following five disjoint subsets:

$(\mathtt{C}^T)$	Constants for *terms*,	$(\mathtt{C}^S)$	Constants for *sets*,
$(\mathtt{C}^N)$	Constants for *nouns*,	$(\mathtt{C}^A)$	Constants for *adjectives*,
$(\mathtt{C}^P)$	Constants for *statements*		

A constant is always followed by a *parameter list*. We denote this as $\mathtt{C}(\overrightarrow{\mathcal{P}})$. This list has for each constant a fixed length ≥ 0, the *arity* of the constant. Parameters $\mathcal{P}$ are either terms, sets or statements: $\mathcal{P} = T|\mathbb{S}|P$.

Examples of constants of WTT

($\mathtt{C}^T$) Constants for $terms$ with parameter lists:
π, the centre of C, $3+6$, the arithmetic mean of 3 and 6, $d(x,y)$, ∇f.

The constants are: π, *the centre*, $+$, *the arithmetic mean*, d and ∇.

The parameter lists are: $(\ )$, (C), $(3,6)$, $(3,6)$, (x,y) and (f), resp.

($\mathtt{C}^S$) Constants for $sets$ with parameter lists: $\mathbb{N}$, A^{c}, $V \to W$, $A \cup B$.

(where A^{c} is the complement of A). The constants are: $\mathbb{N}$, ${}^{\mathsf{c}}$, $\to$, $\cup$.

The parameter lists are: $(\ )$, (A), (V,W), (A,B).

($\mathtt{C}^N$) Constants for $nouns$ with parameter lists: a triangle,
an eigenvalue of A, an edge of $\triangle ABC$, a reflection of V with respect to l.

The constants are: *a triangle*, *an eigenvalue*, *an edge*, *a reflection*.

The parameter lists are: $(\ )$, (A), $(\triangle ABC)$, (V, l).

($\mathtt{C}^A$) Constants for $\textit{adjectives}$ with parameter lists: prime, surjective, Abelian, $\text{continuous on } [a, b]$.

The constants are: prime, surjective, Abelian, continuous.

The parameter lists are: $(\ )$, $(\ )$, $(\ )$, $([a, b])$.

($\mathtt{C}^P$) Constants for $\textit{statements}$ with parameter lists:
$P \text{ lies between } Q \text{ and } R$, $5 \geq 3$, $p \wedge q$, $\neg\forall_{x\in\mathbb{N}}(x > 0)$.

The constants are: *lies between*, $\geq$, $\wedge$, $\neg$.

The parameter lists are: (P, Q, R), $(5, 3)$, (p, q), $(\forall_{x\in\mathbb{N}}(x > 0))$.[2]

[2]Note that the parameters in parameter lists are either $\textit{terms}$ or $\textit{sets}$. Only in the case of statements the parameters may be $\textit{statements}$ as well, as is shown in the last two examples.

Two special constants $\uparrow$ and $\downarrow$ of WTT

$\uparrow$ *lifts* a noun to the corresponding set, $\downarrow$ does the opposite.

Here are examples of these constants:

($\mathtt{C}^S$) $(\text{a natural number})\!\uparrow = \mathbb{N}$, $(\text{a divisor of } 4)\!\uparrow = \{1,2,4\}$,[3]
$(\mathtt{Noun}_{x\in\mathbb{R}}(x > 5))\!\uparrow = \mathtt{Set}_{x\in\mathbb{R}}(x > 5)$.

($\mathtt{C}^N$) $\mathbb{Z}\!\downarrow$ is *an integer*, $(\mathtt{Set}_{x\in\mathbb{R}^2}(|x| = 1))\!\downarrow$ is $\mathtt{Noun}_{x\in\mathbb{R}^2}(|x| = 1)$ or *a point on the unit circle*.

[3]Here again, we used sugaring. We write, $\{1,2,4\}$ for $\mathtt{Set}_{n\in\mathbb{N}}(n = 1 \vee n = 2 \vee n = 4)$. However, the notation with $\mathtt{Set}$ is the only *official* $\mathrm{W\scriptstyle{TT}}$-format.

Binders of WTT

$\mathtt{B} = \mathtt{B}^T|\mathtt{B}^S|\mathtt{B}^N|\mathtt{B}^A|\mathtt{B}^P$ where:

($\mathtt{B}^T$) Binders giving *terms*,
($\mathtt{B}^S$) Binders giving *sets*,
($\mathtt{B}^A$) Binders giving *adjectives*,
($\mathtt{B}^P$) Binders giving *statements*,
($\mathtt{B}^N$) Binders giving *nouns*,

In $\mathtt{B}_{\mathcal{Z}}(\mathcal{E})$, the body $\mathcal{E}$ is one of four categories $\mathcal{E} = T|\mathbb{S}|\mathcal{N}|P$.

Examples:

- $\mathtt{B}^T_{\mathcal{Z}}(\mathcal{E}) = \min_{\mathcal{Z}}(T)|\sum_{\mathcal{Z}}(T)|\lim_{\mathcal{Z}}(T)|\int_{\mathcal{Z}}(T)|\lambda_{\mathcal{Z}}(T)|\lambda_{\mathcal{Z}}(\mathbb{S})|\iota_{\mathcal{Z}}(P)|\ldots$
- $\mathtt{B}^S_{\mathcal{Z}}(\mathcal{E}) = \mathtt{Set}_{\mathcal{Z}}(P)|\bigcup_{\mathcal{Z}}(\mathbb{S})|\iota_{\mathcal{Z}}(P)|\ldots$
- $\mathtt{B}^N_{\mathcal{Z}}(\mathcal{E}) = \mathtt{Noun}_{\mathcal{Z}}(P)|\mathtt{Abst}_{\mathcal{Z}}(T)|\mathtt{Abst}_{\mathcal{Z}}(\mathbb{S})|\mathtt{Abst}_{\mathcal{Z}}(\mathcal{N})|\ldots$
- $\mathtt{B}^A_{\mathcal{Z}}(\mathcal{E}) = \mathtt{Adj}_{\mathcal{Z}}(P)|\ldots$
- $\mathtt{B}^P_{\mathcal{Z}}(\mathcal{E}) = \forall_{\mathcal{Z}}(P)|\ldots$

The λ-binder of WTT

The format of an expression bound by Church's λ-binder is: $\lambda_{\mathcal{Z}}(T/\mathbb{S})$. Here $\lambda_{\mathcal{Z}}(T)$ is a term-valued function and $\lambda_{\mathcal{Z}}(\mathbb{S})$ is a set-valued function. Examples:

$(\mathcal{E} \equiv T)$ The term $\lambda_{x\in\mathbb{R}}(x^2)$ denotes the squaring function on the reals.

$(\mathcal{E} \equiv \mathbb{S})$ The term $\lambda_{n\in\mathbb{N}}\mathtt{Set}_{k\in\mathbb{N}}(k \leq n)$ sends a natural number n to the set $\{0, 1, \ldots, n\}$.

The ι-binder of WTT

Russell's ι is used for a *definite description*: *the* such and such, such that . . . . The general format for an expression bound with the ι-binder is: $\iota_{\mathcal{Z}}(P)$. The result of the binding of a sentence by means of ι can either be a term or a set (therefore we find $\iota_{\mathcal{Z}}(P)$ both in the $\mathtt{B}^T$- and in the $\mathtt{B}^S$-list). For example:

- The term $\iota_{n\in\mathbb{N}}(2 < n < \pi)$ describes natural number 3.
- The set $\iota_{U:\,\textsf{SET}}\,(3 \in U \wedge |U| = 1)$ describes the singleton set $\{3\}$ (or $Set_{n\in\mathbb{N}}(n = 3)$ in unsugared format). (The declaration U : SET expresses that U *is a set*.)

The $\texttt{Noun}$-binder of WTT

Next to set comprehension, we allow $noun\ comprehension$, i.e. the construction of a noun.

For noun comprehension we introduce the binder $\texttt{Noun}$. It is used for an $indefinite\ description$: *a such and such, such that ....*

Hence, the general format of a phrase with $\texttt{Noun}$-binder is: $\texttt{Noun}_{\mathcal{Z}}(P)$, i.e. *a noun saying of $\mathcal{Z}$ that P*.

Examples:

- The noun $\texttt{Noun}_{x\in\mathbb{R}}(5 < x < 10)$ is *a real number between 5 and 10*.
- $\texttt{Noun}_{V:\ \mathsf{SET}}\ (|V| = 2)$ is *a set with two elements*.

The `Abst`-binder of WTT

The `Abst`-binder $abstracts$ from a term T, a set $\mathbb{S}$ or a noun $\mathcal{N}$ and delivers a noun. It is the formal counterpart of the modifier *for some . . .* . One may read $\texttt{Abst}_{\mathcal{Z}}(T/\mathbb{S}/\mathcal{N})$ as *a term T, or a set $\mathbb{S}$, or a noun $\mathcal{N}$, for some $\mathcal{Z}$*.

Here are examples of the three kinds of nouns $\texttt{Abst}_{\mathcal{Z}}(T/\mathbb{S}/\mathcal{N})$:

($\mathcal{E} \equiv T$) $\texttt{Abst}_{n\in\mathbb{N}}(n^2)$ represents *a term n^2 $for\ some$ natural number n*, i.e. *the square of some natural number*.

($\mathcal{E} \equiv \mathbb{S}$) $\texttt{Abst}_{n\in\mathbb{N}}\texttt{Set}_{x\in\mathbb{R}}(x > n)$ represents *a set $\{x \in \mathbb{R}|x > n\}$ $for\ some$ natural number n*, i.e. *an interval of the form (n, ∞)*, with $n \in \mathbb{N}$.

($\mathcal{E} \equiv \mathcal{N}$) $\texttt{Abst}_{n\in\mathbb{N}}\texttt{Noun}_{x\in\mathbb{R}}(10n \leq x < 10n + 1)$ represents *a real number in the interval $[10n, 10n + 1)$ $for\ some$ n*, i.e. *a non-negative real number which, written in decimal notation, has a zero at the position just before the decimal point*.

The `Adj`-binder of WTT

- Adjectives can be constructed with the `Adj`-binder.
- One can read $\mathtt{Adj}_{\mathcal{Z}}(P)$ as: *the adjective saying of* $\mathcal{Z}$ *that* P.
- E.g.: $\mathtt{Adj}_{n\in\mathbb{N}}(\exists_{k\in\mathbb{N}}(n = k^2 + 1))$ is an adjective saying of a natural number that it is a square plus 1.
- One could give this adjective a name, say *oversquare* and hence say things like *5 is oversquare* or *Let m be an oversquare number*.

Phrases of WTT

Phrases can be terms, sets, nouns or adjectives:

$$
\begin{array}{ll}
\mathcal{T} = \mathtt{C}^{\mathcal{T}}(\overrightarrow{\mathcal{P}})|\mathtt{B}^{\mathcal{T}}_{\mathcal{Z}}(\mathcal{E})|\mathtt{V}^{\mathcal{T}} & \mathbb{S} = \mathtt{C}^{S}(\overrightarrow{\mathcal{P}})|\mathtt{B}^{S}_{\mathcal{Z}}(\mathcal{E})|\mathtt{V}^{S} \\
\mathcal{N} = \mathtt{C}^{\mathcal{N}}(\overrightarrow{\mathcal{P}})|\mathtt{B}^{\mathcal{N}}_{\mathcal{Z}}(\mathcal{E})|\mathcal{A}\mathcal{N} & \mathcal{A} = \mathtt{C}^{\mathcal{A}}(\overrightarrow{\mathcal{P}})|\mathtt{B}^{\mathcal{A}}_{\mathcal{Z}}(\mathcal{E}).
\end{array}
$$

We already gave examples of $\mathtt{C}^{T}(\overrightarrow{\mathcal{P}})$, $\mathtt{C}^{S}(\overrightarrow{\mathcal{P}})$, $\mathtt{C}^{N}(\overrightarrow{\mathcal{P}})$ and $\mathtt{C}^{A}(\overrightarrow{\mathcal{P}})$ and of $\mathtt{B}^{T}_{\mathcal{Z}}(\mathcal{E})$, $\mathtt{B}^{S}_{\mathcal{Z}}(\mathcal{E})$, $\mathtt{B}^{N}_{\mathcal{Z}}(\mathcal{E})$ and $\mathtt{B}^{A}_{\mathcal{Z}}(\mathcal{E})$.

The combination $\mathcal{A}\mathcal{N}$ gives a (new) noun which is a combination of an adjective and a noun. E.g.: *isosceles triangle*, *convergent series*.

Statements of WTT

Abstract syntax for the category of *statements* is: $P = \mathtt{C}^P(\overrightarrow{\mathcal{P}})|\mathtt{B}^P_{\mathcal{Z}}(\mathcal{E})|\mathtt{V}^P.$

Examples of $\mathtt{C}^P(\overrightarrow{\mathcal{P}})$ and of $\mathtt{B}^P_{\mathcal{Z}}(\mathcal{E})$ (with the $\forall$-binder for $\mathtt{B}^P$) were already given.

The abstract syntax for the set $\mathbf{T}$ of typing statements ($\mathbf{T} \subseteq P$) is:
$\mathbf{T} = \mathbb{S} : \text{SET} |\mathcal{S} : \text{STAT} |T : \mathbb{S}|T : \mathcal{N}|T : \mathcal{A}.$

Examples of these cases include: $\mathtt{Set}_{n\in\mathbb{N}}(n \leq 2)$: *SET* , $p \wedge q$: *STAT* , $3 \in \mathbb{N}$,[4] AB : *an edge of* $\triangle ABC$, $\lambda_{x\in\mathbb{R}}(x^2)$: *differentiable*.

[4] As this example shows, we often replace $t : s$ by $t \in s$, with abuse of notation.

Definitions of WTT

- The category $\mathcal{D} = \mathcal{D}^{\varphi}|\mathcal{D}^{P}$ of *definitions* introduces new constants.
- We distinguish between *phrase definitions* $\mathcal{D}^{\varphi}$ and *statement definitions* $\mathcal{D}^{P}$.
- Phrase definitions fix a constant representing a phrase.
- Statement definitions introduce a constant embedded in a statement.
- In definitions, the defined constant is separated from the phrase or statement it represents by the symbol ":=".

Phrase definitions of WTT

We take $\mathcal{D}^\varphi \quad = \quad \mathtt{C}^T(\overrightarrow{V}) := T \mid \mathtt{C}^S(\overrightarrow{V}) := \mathbb{S} \mid \mathtt{C}^N(\overrightarrow{V}) := \mathcal{N} \mid \mathtt{C}^{\mathcal{A}}(\overrightarrow{\mathcal{V}}) := \mathcal{A}$

Examples of phrase definitions are:

$(\mathtt{C} \equiv \mathtt{C}^T)$ $\textit{the arithmetic mean}$ of a and b $:= \iota_{z\in\mathbb{R}}(z = \frac{1}{2}(a+b))$,

$(\mathtt{C} \equiv \mathtt{C}^{\mathbb{S}})$ $\mathbb{R}^+ := \mathtt{Set}_{x\in\mathbb{R}}(x > 0)$,

$(\mathtt{C} \equiv \mathtt{C}^{\mathcal{N}})$ $\textit{a unit}$ of G with respect to $\cdot := \mathtt{Noun}_{e\in G}(\forall_{a\in G}(a \cdot e = e \cdot a = a))$

$(\mathtt{C} \equiv \mathtt{C}^{\mathcal{A}})$ $\textit{prime} := \mathtt{Adj}_{n\in\mathbb{N}}(n > 1 \wedge \forall_{k,l\in\mathbb{N}}(n = k \cdot l \Rightarrow k = 1 \vee l = 1))$.

The variable lists in the four examples are: (a, b), $(\ )$, $(G, \cdot)$, $(\ )$. These variables must be introduced (*declared*) in a context.

For the first definition, such a context can be e.g. $a : \mathbb{R}, b : \mathbb{R}$.

For the third definition the context is: $G :$ SET , $\cdot : G \to G$.

Statement definitions of WTT

$\mathcal{D}^P \;=\; \mathtt{C}^P(\vec{V}) := P$ is the category of statement definitions *defining constant* $\mathtt{C}^P$.

For example in a context like: *Let a and b be lines*:

$(\mathtt{C} \equiv \mathtt{C}^P)\;\; a \textit{ is parallel to } b \;:=\; \neg\exists_{P:\text{ a point}}(P \text{ lies on } a \wedge P \text{ lies on } b).$

Contexts of WTT

A *context* Γ is a list of declarations $\mathcal{Z}$ and statements P:

$$\mathbb{F} \quad = \quad \emptyset \mid \mathbb{F}, \mathcal{Z} \mid \mathbb{F}, P.$$

A declaration in a context represents the *introduction of a variable* of a known type.

A statement in a context stands for an *assumption*.

Lines of WTT

A *line* l contains either a statement or a definition, relative to a context:

$$\mathbf{l} \;=\; \mathbb{\Gamma} \triangleright P \mid \mathbb{\Gamma} \triangleright \mathcal{D}.$$

The symbol $\triangleright$ is a separation marker between the context and the statement or definition.

Here are two examples of lines:

A statement line: $\quad x:\mathbb{N}, y:\mathbb{N}, x<y \;\triangleright\; x^2<y^2$,

A definition line: $\quad x:\mathbb{R}, x>0 \;\triangleright\; ln(x) := \iota_{y\in\mathbb{R}}(e^y = x)$.

Books of WTT

A *book* B is a list of lines: $\quad \mathbf{B} \;=\; \emptyset \mid \mathbf{B} \circ \mathbf{l}.$

A simple example of a book consisting of two lines is the following:

$x : \mathbb{R}, x > 0 \;\rhd\; ln(x) := \iota_{y \in \mathbb{R}}(e^y = x) \;\; \circ$

$\emptyset \;\rhd\; ln(e^3) = 3\;.$

MathLang's Grammatical categories

They extend those of WTT with blocks and flags.

- $\mathtt{T}$ terms
- $\mathtt{S}$ sets
- $\mathtt{N}$ nouns
- $\mathtt{A}$ adjectives
- $\mathtt{P}$ statements
- $\mathtt{D}$ definitions
- $\mathtt{Z}$ declarations
- Γ contexts with flags
- $\mathtt{L}$ lines
- $\mathtt{K}$ blocks
- $\mathtt{B}$ books

The Grammatical Categories of MathLang

$$\underbrace{\underbrace{\underbrace{\text{Let } \underbrace{\mathfrak{M}}_{\mathtt{S}} \text{ be the set of } \ldots,}_{\mathtt{Z}} \;\; \underbrace{\underbrace{y}_{\mathtt{T}} \;\; \underbrace{\text{a natural number},\ldots}_{\mathtt{N}}}_{\mathtt{Z}} \;\; \text{if } \underbrace{\underbrace{x}_{\mathtt{T}} \text{ belongs to } \underbrace{\mathfrak{M}}_{\mathtt{S}}}_{\mathtt{P}}}_{\Gamma} \quad \text{then } \underbrace{\underbrace{\underbrace{x}_{\mathtt{T}} + \underbrace{y}_{\mathtt{T}}}_{\mathtt{T}} = \underbrace{\underbrace{y}_{\mathtt{T}} + \underbrace{x}_{\mathtt{T}}}_{\mathtt{T}}}_{\mathtt{P}}}_{\mathtt{L}}$$

Figure 4: A mathematical line and its grammatical categories

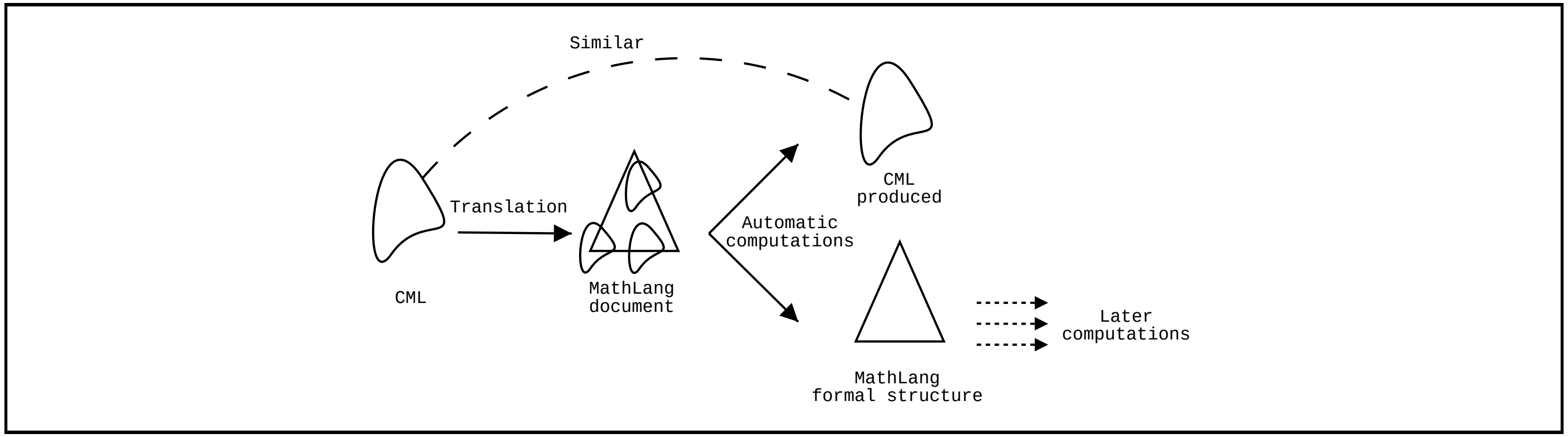


Figure 5: Translation process of MathLang

Derivation rules of WTT

(1) B is a weakly well-typed book: $\vdash B :: \mathbf{B}$.

(2) Γ is a weakly well-typed context relative to book B: $B \vdash \Gamma :: \mathbb{F}$.

(3) t is a weakly well-typed term, etc., relative to book B and context Γ:

$$\begin{array}{lll} B;\Gamma \vdash t :: T, & B;\Gamma \vdash s :: S, & B;\Gamma \vdash n :: N, \\ B;\Gamma \vdash a :: A, & B;\Gamma \vdash p :: P, & B;\Gamma \vdash d :: D \end{array}$$

$OK(B;\Gamma)$. stands for: $\vdash B :: \mathbf{B}$, *and* $B \vdash \Gamma :: \mathbb{F}$

A preface for a book B could look like:

constant name	weak type	constant name	weak type
$\mathbb{R}$	S	$\cup$	$S \times S \to S$
$\sqrt{}$	$T \to T$	$\geq$	$T \times T \to P$
$+$	$T \times T \to T$	$\wedge$	$P \times P \to P$

- $\mathbb{R}$ has no parameters and is a set.
- $\sqrt{}$ is a constant with one parameter, a term, delivering a term.
- $\geq$ is a constant with two parameters, terms, delivering a statement.
- $\texttt{prefcons}(B) = \{\mathbb{R}, \sqrt{}, +, \cup, \geq, \wedge\}$.

- $\mathtt{dvar}(\emptyset) = \emptyset \qquad \mathtt{dvar}(\Gamma', x : W) = \mathtt{dvar}(\Gamma'), x \qquad \mathtt{dvar}(\Gamma', P) = \mathtt{dvar}(\Gamma')$

$$\frac{OK(B;\Gamma), \quad x \in \mathtt{V}^{\mathtt{T/S/P}}, \quad x \in \mathtt{dvar}(\Gamma)}{B;\Gamma \ \vdash \ x :: T/S/P} \quad (var)$$

$$\frac{B;\Gamma \ \vdash \ n :: N, \quad B;\Gamma \ \vdash \ a :: A}{B;\Gamma \ \vdash \ an :: N} \quad (adj{-}noun)$$

$$\frac{}{\vdash \ \emptyset :: \mathbf{B}} \quad (emp{-}book)$$

$$\frac{B;\Gamma \ \vdash \ p :: P}{\vdash \ B \circ \Gamma \triangleright p :: \mathbf{B}} \qquad \frac{B;\Gamma \ \vdash \ d :: D}{\vdash \ B \circ \Gamma \triangleright d :: \mathbf{B}} \quad (book{-}ext)$$

Example in WTT

CML: the square root of the third power of a natural number

WTT: $\mathtt{Abst}_{n:\mathbb{N}}(\sqrt{n^3})$

The preface is:

	constant name	weak type
(i)	^{3}	$T \to T$
(ii)	$\sqrt{}$	$T \to T$
(iii)	$\mathbb{N}$	S
(iv)	$\mathtt{Abst}$	$T \to N$

The categories are:

subexp	category	subexp	category	subexp	category
n	T	n	T	$\mathtt{Abst}_{n:\mathbb{N}}(\sqrt{n^3})$	N
n^3	T	$\mathbb{N}$	S		
$\sqrt{n^3}$	T	$n : \mathbb{N}$	$\mathcal{Z}$		

We need to derive $B; \Gamma \vdash \mathtt{Abst}_{n:\mathbb{N}}(\sqrt{n^3}) :: N$ for some B and Γ.

But it is clear that $B = \Gamma = \emptyset$.

Example in WTT

(1)		$\vdash$	$\emptyset$ :: $\mathbf{B}$	$(emp{-}book)$
(2)	$\emptyset$	$\vdash$	$\emptyset$ :: $\mathbb{I}$	$(emp{-}cont, 1)$
(3)	$\emptyset; \emptyset$	$\vdash$	$\mathbb{N}$ :: S	$(ext{-}cons, 1, 2, iii)$
(4)	$\emptyset$	$\vdash$	$n : \mathbb{N}$ :: $\mathbb{I}$	$(term{-}decl, 1, 2, 3, *)$
(5)	$\emptyset; n : \mathbb{N}$	$\vdash$	n :: T	$(var, 1, 4, *)$
(6)	$\emptyset; n : \mathbb{N}$	$\vdash$	n^3 :: T	$(ext{-}cons, 1, 4, i, 5)$
(7)	$\emptyset; n : \mathbb{N}$	$\vdash$	$\sqrt{n^3}$ :: T	$(ext{-}cons, 1, 4, ii, 6)$
(8)	$\emptyset; \emptyset$	$\vdash$	$\mathtt{Abst}_{n:\mathbb{N}}(\sqrt{n^3})$ :: N	$(bind, 1, 4, iv, 7)$

Figure 6: Derivation that $\mathtt{Abst}_{n:\mathbb{N}}(\sqrt{n^3})$ is a noun

Example 2 in WTT

Our second example concerns a text with a definition and its application:

DEFINITION A *Fermat-sum* is a natural number which is the sum of two squares of natural numbers.

LEMMA The product of a square and a Fermat-sum is a Fermat-sum.

A WTT-translation could be the following small WTT-book B of two lines (both with an an empty context), one a definition and the other a statement. So the abstract format of B is: $\emptyset \rhd D \ \circ \ \emptyset \rhd S$:

$$a \ Fermat\text{-}sum \ := \mathtt{Noun}_{n \in \mathbb{N}} \exists_{k \in \mathbb{N}} \exists_{l \in \mathbb{N}} (n = k^2 + l^2)$$

$$\forall_{u: \ a \ square} \forall_{v: \ a \ Fermat\text{-}sum} (uv : \ \ a \ Fermat\text{-}sum)$$

Example 2 in MathLang

The original CML text is given by figure 7. Our translation of this text into MathLang is shown in figure 8. Figure 9 is the CML output we obtain from this encoding.

Definition 2. A *Fermat-sum* is a natural number which is the sum of two squares of natural numbers.

Lemma 3. The product of a square and a Fermat-sum is a Fermat sum.

Figure 7: *Fermat-sum* example: original text

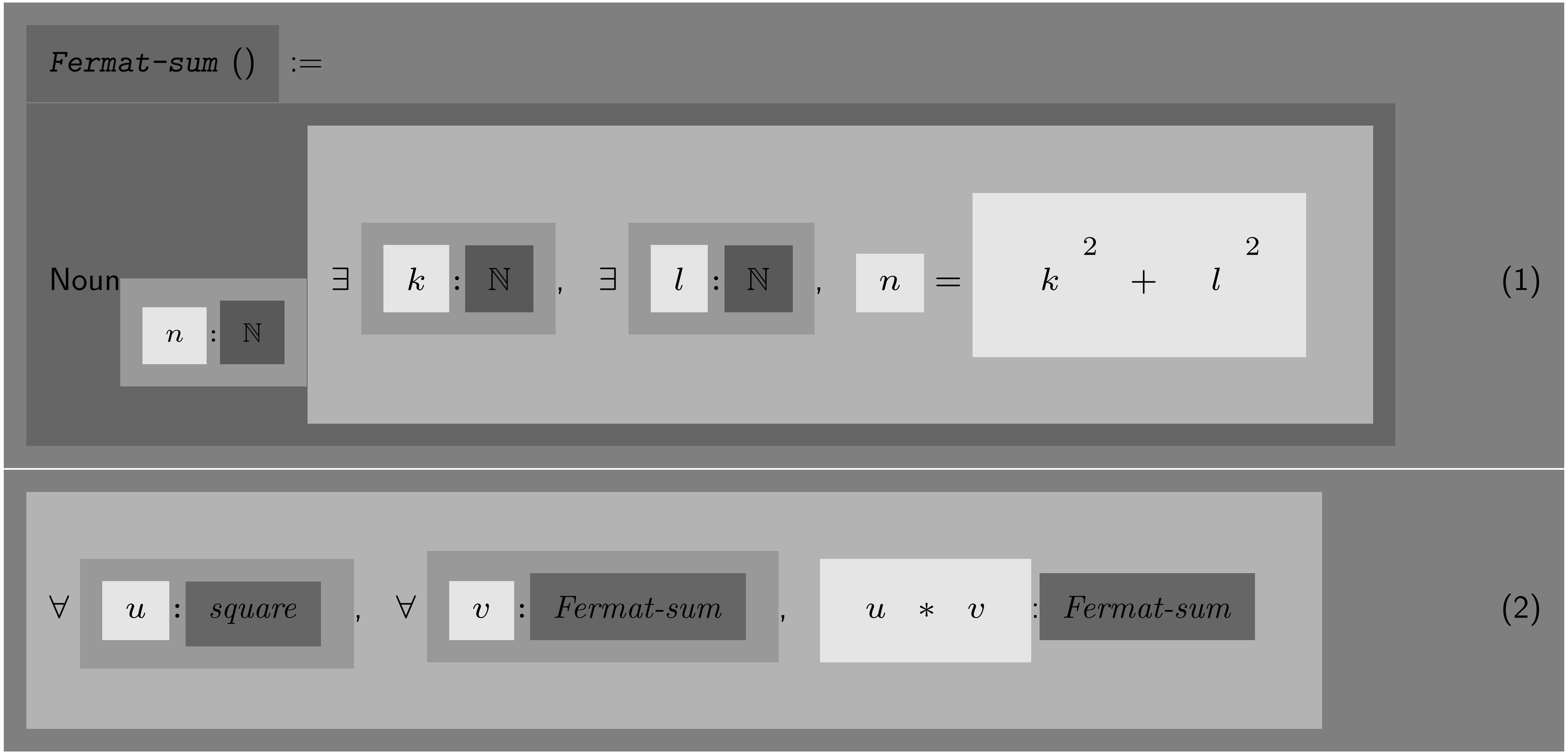


Figure 8: *Fermat-sum* example: symbolic structural view of MathLang

Definition 4. [Fermat-sum]

A *Fermat-sum* is

a natural number which is the sum of two squares of natural numbers . 1

Lemma 5.

The product of a *square* and a *Fermat-sum* is a *Fermat-sum* . 2

Figure 9: *Fermat-sum* example: CML view of MathLang

Comparaison with other work

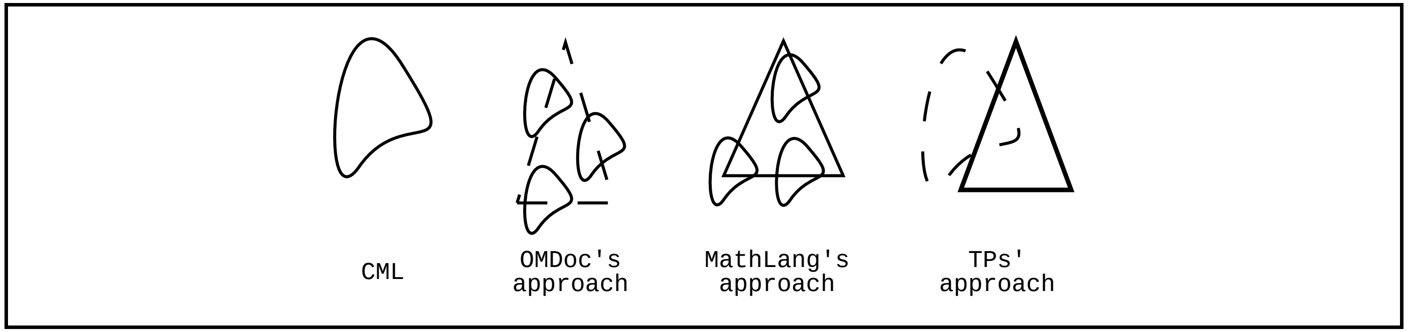


Figure 10: Approaches

- The formalisation of a language of mathematics should separate the questions:
  - *which type theory is necessary for which part of mathematics*
  - *which language should mathematics be written in*.
- Mathematicians don't usually know or work with type theories.
- Mathematicians usually *do* mathematics (manipulations, calculations, etc), but are not interested in general in reasoning *about* mathematics.

Another MathLang example

T Terms | S Sets | N Nouns | P Statements | Z Declarations | Γ Context

Let $\mathfrak{M}$ be a set ,
y and x are natural numbers ,
if x belongs to $\mathfrak{M}$

then $x + y = y + x$

MathLang Checking

T Terms | S Sets | N Nouns | P Statements | Z Declarations | Γ Context

Let $\mathfrak{M}$ be a set ,

y and x are natural numbers ,

if x belongs to $\mathfrak{M}$

then $x + y$ $\Leftarrow$ **error**

Another MathLang example

blocks flags references

Theorem 6. [Commutative Law of Addition]

$$x + y = y + x.$$

Proof Fix y, and $\mathfrak{M}$ be the set of all x for which the assertion holds.

I) We have

$$y + 1 = y',$$

and furthermore, by the construction in the proof of Theorem 4,

$$1 + y = y',$$

so that

$$1 + y = y + 1$$

and 1 belongs to $\mathfrak{M}$.

II) If x belongs to $\mathfrak{M}$, then

$$x + y = y + x,$$

Therefore

$$(x + y)' = (y + x)' = y + x'.$$

By the construction in the proof of Theorem 4, we have

$$x' + y = (x + y)',$$

hence

$$x' + y = y + x',$$

so that x' belongs to $\mathfrak{M}$. The assertion therefore holds for all x.

MathLang skeleton

$x : \mathbb{N},\ y : \mathbb{N} \rhd \mathsf{Th6}(x, y) := x + y = y + x$ (97)

Proof Theorem 6 {2.5.4}

Proof Theorem 6 part I {2.5.4.1}

$\boxed{y : \mathbb{N}}$

$\boxed{\mathfrak{M} : \mathsf{SET}}$

$\boxed{\forall_{x:\mathfrak{M}} \mathsf{Th6}(x, y)}$

$(\mathsf{Def}\ +(38)) \rhd y + 1 = y'$ (98)

$\{2.5.1\} \rhd 1 + y = y'$ (99)

$(98), (99) \rhd 1 + y = y + 1$ (100)

$(100) \rhd \mathsf{Th6}(1, y)$ (101)

$(101) \rhd 1 : \mathfrak{M}$ (102)

Proof Theorem 6 part II {2.5.4.2}

$\boxed{x : \mathfrak{M}}$

$\mathsf{Th6}(x, y) \rhd x + y = y + x$ (103)

$(103) \rhd (x + y)' = (y + x)'$ (104)

$(\mathsf{Def}\ +(39)) \rhd (y + x)' = y + x'$ (105)

$(104), (105) \rhd (x + y)' = y + x'$ (106)

$\{2.5.2\} \rhd x' + y = (x + y)'$ (107)

$(107), (\mathsf{Def}\ +(39)) \rhd x' + y = y + x'$ (108)

$(108) \rhd \mathsf{Th6}(x', y)$ (109)

$(109) \rhd x' : \mathfrak{M}$ (110)

$\mathsf{Ax5}(\mathfrak{M}, (102), (110)) \rhd \mathbb{N} \subset \mathfrak{M}$ (111)

$(111) \rhd \forall_{x:\mathbb{N}} \forall_{y:\mathbb{N}} \mathsf{Th6}(x, y)$ (112)

References

Alonzo Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5:56–68, 1940b.

H. B. Curry and R. Feys. *Combinatory Logic I*. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1958a.

G. Frege. Letter to Russell. English translation in Heijenoort [1967], pages 127–128, 1902.

T.L. Heath. *The Thirteen Books of Euclid's Elements*. Dover Publications, Inc., New York, 1956.

J. van Heijenoort, editor. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*. Harvard University Press, Cambridge, Massachusetts, 1967.

A. Heyting. *Mathematische Grundlagenforschung. Intuitionismus. Beweistheorie.* Ergebnisse der Mathematik und ihrer Grenzgebiete. Springer-Verlag, Berlin, 1934a.

W. A. Howard. The formulaes-as-types notion of construction. In J. R[oger] Hindley and J[onathan] P. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980a. ISBN 0-12-349050-2. An earlier version was privately circulated in 1969.

F. Kamareddine. Reviewing the classical and the de Bruijn notation for λ-calculus and Pure Type Systems. *J. Logic Comput.*, 11(3):363–394, 2001.

F. Kamareddine, L. Laan, and R. P. Nederpelt. Types in logic and mathematics before 1940. *Bulletin of Symbolic Logic*, 8(2):185–245, June 2002.

F. Kamareddine, T. Laan, and R. Nederpelt. Automath and pure type systems. In *Thirty five years of automating mathematics*, volume 28 of *Applied Logic Series*, pages 71–123. Kluwer, 2003.

F. Kamaredine, M Maarek, and J.B. Wells. Flexible encoding of mathematics on the computer. 2004a.

F. Kamaredine, M Maarek, and J.B. Wells. Mathlang: An experience driven language of mathematics. *Electronic Notes in Theoretical Computer Science 93C*, pages 123–145, 2004b.

F. Kamaredine and R.P. Nederpelt. A refinement of de bruijn's formal language of mathematics. *Logic, Language and Information*, 2004.

A. N. Kolmogorov. Zur Deutung der Intuitionistischen Logik. *Mathematisches Zeitschrift*, 35:58–65, 1932a.

Edmund Landau. *Foundations of Analysis*. Chelsea, 1951.

S. MacLane. *Categories for the Working Mathematician*. pringer, 1972.

B. Russell. Letter to Frege. English translation in Heijenoort [1967], pages 124–125, 1902.

B. Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30:222–262, 1908. Also in Heijenoort [1967], pages 150–182.

W.W. Tait. Infinitely long terms of transfinite type. In J.N. Crossley and M.A.E. Dummett, editors, *Formal Systems and Recursive Functions*, Amsterdam, 1965. North-Holland.

A.N. Whitehead and B. Russell. *Principia Mathematica*, volume I, II, III. Cambridge University Press, 1910^1, 1927^2. All references are to the first volume, unless otherwise stated.

E. Zermelo. Untersuchungen über die Grundlagen der Mengenlehre. *Math. Annalen*, 65:261–281, 1908.