

# A Logically Saturated Extension of $\bar{\lambda}\mu\tilde{\mu}$

Lionel Elie Mamane, Herman Geuvers, and James McKinna

Institute for Computing and Information Sciences  
Radboud University Nijmegen  
lionel@mamane.lu, herman@cs.ru.nl and james@cs.ru.nl

**Abstract** This paper presents a proof language based on the work of Sacerdoti Coen [1,2], Kirchner [3] and Autexier [4] on  $\bar{\lambda}\mu\tilde{\mu}$ , a calculus introduced by Curien and Herbelin [5,6]. Just as  $\bar{\lambda}\mu\tilde{\mu}$  preserves several proof structures that are identified by the  $\lambda$ -calculus, the proof language presented here aims to preserve as much proof structure as reasonable; we call that property being *logically saturated*. This leads to several advantages when the language is used as a generic exchange language for proofs, as well as for other uses.

We equip the calculus with a simple rendering in pseudo-natural language that aims to give people tools to read, understand and exchange terms of the language. We show how this rendering can, at the cost of some more complexity, be made to produce text that is more natural and idiomatic, or in the style of a declarative proof language like Isar or Mizar.

## 1 Introduction

A central need of Mathematical Knowledge Management is languages for representation of mathematical documents and objects and discourse within them. We here focus on proofs, at a level more aware of the logic reasoning than efforts such as OMDoc. From our focus on proofs, follows a focus the part of the MKM toolchain that deals with proofs, namely proof assistants. We aim at a proof language that is general enough to capture different notions of proofs, that captures the structure of a proof in detail; a language that differentiates distinct proofs, but identifies two texts that represent the same proof. Such a proof language can be used as a common ground for interchange between different systems, as a language to speak about proofs and transformations thereof (e.g. automatic proof enhancement, rendering into natural language, ...).

Because validity of proofs is more closely tied to the precise semantics than the informal meaning of an expression, we find that the best level to address interchange of proofs is not the content level (using the terminology of [7]), but something in between the content level and the semantic level.

Another requirement of such a proof language would be to have a nice natural language-style pretty-printing; the latter transformation ideally being simple enough to be done in one's head, so that a term in that language be readable by itself for someone that knows the language. In this regard, the natural language transformation of  $\bar{\lambda}\mu\tilde{\mu}$  in [1] is very attractive: the transformation is purely

structural, and the term is read strictly from left to right. However, it does not satisfyingly treat the whole calculus and  $\bar{\lambda}\mu\tilde{\mu}$  (as extended to predicate logic in Fellowship [3]) still identifies proofs we'd like to differentiate.

This paper presents a more discerning extension of  $\bar{\lambda}\mu\tilde{\mu}$  and a basic rendering of it in pseudo-natural language. We show how the rendering can be enhanced to produce text that is more pleasing to read, and sketch how  $\bar{\lambda}\mu\tilde{\mu}$  can be translated into the input language of proof assistants. The language presented here covers only implicational logic (with an extension to full propositional logic given in the appendix and in [8]) and only proofs where every step taken is an atomic step of reasoning. So, seen from the viewpoint of proof assistants, we only deal with proofs where no automation is used. (Or, alternatively, the proof that is constructed by the automation, as opposed to the proof written by the user.) Naturally, the language will be extended in future work to address this limitation.

### 1.1 What Is a Proof?

The traditional answer from a logician's point of view is that it is a derivation in a formal system of rules, in the form of a DAG / tree / lambda-term. From a more general mathematician point of view, it is a text that convinces his peers, for example a text that convinces them that if they would spend the effort and time to write the fully formal derivation, they would get such a derivation.

With our view centred on proof assistants, we consider the user input to the proof assistant, that which gets stored in the library of the proof assistant, to be a good candidate for the right notion of proof. Indeed, if the automation changes and finds a different "logician's proof" for the same input, the user won't consider that the proof he has written changed. A good test for the suitability of a candidate proof format is thus how well it captures these "proof assistant proofs".

This notion of proof is both coarser and finer than logician's proofs:

- it is coarser, because it glosses over automation done by the proof assistant; if the automation procedure changes, and finds a different logician's proof of a step done by automation, we still consider it the same proof
- it is finer, because it separates cases where the same logician's proof (e.g.  $\lambda$ -term) is produced in different ways, e.g. by a top-down proof or by a bottom-up proof, or by a proof that is partially top-down and partially bottom-up.

### 1.2 Design

**Differentiating Power** We already mentioned that we want our proof language to distinguish different proofs, but identify texts that code for the same proof. This naturally begs the question: Given two texts, when do they represent different proofs and when do they represent the same proof?

We want to preserve the *intentional* content of a proof, the story that is being told. For example, a proof that first establishes  $A$ , then  $B$  and from these two concludes  $C$  is not the same as a proof that first established  $B$ , then  $A$

and then concludes. So we want our language to distinguish the order in which things happen, and to distinguish a forward-style (bottom-up) proof from a backward-style (top-down) proof, and to distinguish these from proofs that are done partially forwards and partially backwards.

As we focus on the logical content of proofs, it seems natural that we identify texts that vary only by purely linguistic differences. For example, the proofs at the right differ only linguistically from the corresponding proof at the left:

case 1: $A$ holds ...	either $A$ ...
case 2: $\neg A$ holds ...	or $\neg A$ ...
H: $A$ by B	H: $A$ by B
hence C	thus C by H

But if the difference comes from application of a different reasoning step, a different deduction rule, then it is not the same proof and the language should distinguish them. For example:

we have $A \rightarrow C \wedge B$	we have $A \rightarrow C \wedge B$
in particular we have $A \rightarrow C$	that is, we have $A \rightarrow C$ ( $x$ )
we already established $A$ in lemma 5	and we have $B$ ( $y$ )
thus $C$	by lemma 5 and $x$ , we conclude $C$

The left proof uses a projection (from  $A \wedge B$ , we deduce *only*  $A$ ), while the right proof uses a full decomposition (from  $A \wedge B$ , we deduce *both*  $A$  and  $B$ ), it is not the same deduction rule.

**Saturated System** This leads to a concept in opposition with the traditional design of proof languages, which is to make them *minimal* at the logical level: with the possible exception of the cut rule, remove any deduction rule from the system and it is not complete anymore. E.g. for conjunction, if you have both projections, you don't need full decomposition and vice-versa. We aim for a language that is *saturated*: Any step that an author can reasonably see as an atomic step, as a rule of reasoning that his reader will not doubt, should be a rule of the language. Any deduction rule that a proof assistant, or a logic, can reasonably choose as part of its "minimal set" should be a rule of the language.

- When the language is used as a proof *interchange* language, it allows the language to be neutral towards the choices made by different proof assistants; the language then is not closer to any one specific family of proof assistants than to another. By its saturation, it is close to all of them.

A prime example of this is the implementation of classical logic: Some proof assistants implement intuitionistic logic augmented with an axiom (excluded middle, Peirce's law, double negation law, ...). Others, such as PVS, use the reasoning with multiple goals of sequent calculus, where proving any of the goals (not all of them) finishes the proof. A proof in one style can be transformed into the other style mechanically, but this produces a *different* proof of the same proposition. A language that would provide classical logic

through the double negation rule would naturally get into difficulties when trying to represent PVS proofs faithfully, because it would not be able to represent a PVS proof that makes essential use of multiple goals.

- It makes the language particularly well suited to talk about proof manipulations. For example, to present an algorithm to transform a proof from one system into another system, because it has the concepts and rules from both systems, it gives a way to talk about them and relate them. For example, an algorithm to transform a multiple-goal proof into a double-negation law proof can be expressed as a transformation on terms of the language.
- It gives a language in which to study and characterise what kinds of proofs what system can handle, expressed as sublanguage of the language.
- When the language is used as a proof *authoring* language, it has the advantage to present all choices in a uniform way, without arbitrary distinction between which (in the user’s intuition) atomic step is atomic for the system and which step is a lemma application. There is no reason the user should have to care about that distinction.

Taking all this together, let’s imagine a user that wants to work in classical logic, but is used only to its expression as intuitionistic logic plus double negation law. He ploughs on with his proof, but his proof assistant keeps track of the alternative goals he could be proving instead of the goal he is thinking of, and informing him of that list in a side-window. The user keeps an eye on it, and notices that this other goal seems easier to prove at this point. He does so. He doesn’t understand the proof he has written (that multiple goals stuff is a bit mysterious, that’s not how logic works in his mind), but he asks the system to transform it into an intuitionistic logic plus excluded middle proof, and he has a proof he can read and understand, a proof that he can present to his colleagues (which, being of a similar background, wouldn’t be convinced by a goal-switching proof). By being based on a saturated logical system, the proof assistant has made its user’s life easier.

Naturally, the kernel of the proof assistant, the fundamental proof checking, can (e.g. for De Bruijn criterion reasons) still happen in a minimal system, interpreting the other rules as lemma applications. And the next version of the proof assistant may actually use a different minimal set of rules, and no one will notice. This is a kind of “abstract datatype” approach to logic: One does not need to look into what choices the proof assistant one uses has made; one is free to do the things the logic one works in allows, the system implementing the abstract signature maps some steps to atomic steps and some others to lemmas, but this is none of our concern.

**Understandability** Expressions of the language should *mean* something to a reader, be understandable. This is ensured if the user knows a transformation to natural language that is simple enough that he can do it in his head, if every construct of the language has a clear semantic and maps to a concept or a rule that the reader recognises.

**Flexibility** The language should capture different proof assistants' notion of a proof, but also a human's natural language view of a rigorous proof.

## 2 $\bar{\lambda}\mu\tilde{\mu}$

### 2.1 Syntax

The  $\bar{\lambda}\mu\tilde{\mu}$  calculus, which covers implication logic is made of three interdependent syntactical categories:

**term** an expression of the category  $v$ . A typing judgement for a term is of the form  $\Gamma \vdash v : T \mid \Delta$ . The  $v : T$  part is called the *stoup*. By abuse of language, we say “ $v$  is of type  $T$ ” when  $\Gamma \vdash v : T \mid \Delta$  for some  $\Gamma, \Delta$  that are clear from the context. The typing judgement should be understood as:  $v$  proves the sequent  $\Gamma \vdash T, \Delta$ .  $T$  is singled out as the thesis one is currently working on; switching is allowed, but is an explicit step. The top level of a term is where right introduction rules happen, where one works on the right side of the sequent.

A term variable will be denoted as  $x, y, z, \dots$

The natural language rendering of a term  $v$  thus naturally is some text that is a proof of the sequent that types  $v$ , with the type of  $v$  as focus (current thesis) at the beginning of the text.

**environment** an expression of the category  $e$ . A typing judgement for it is of the form  $\Gamma \mid e : T \vdash \Delta$ . The conventions made for terms apply mutatis mutandis. The typing judgement should be understood as:  $e$  expects (consumes) a proof of  $T$  (a term  $v$  typed by  $\Gamma \vdash v : T \mid \Delta$ ) and continues further with the proof of sequent  $\Gamma, T \vdash \Delta$ , using the  $v$  it has consumed. The top level of an environment is where left introduction rules happen, where one works on the hypotheses or a previously established proposition.

An environment variable will be denoted as  $\alpha, \beta, \gamma, \delta, \dots$

The natural language rendering of an environment thus naturally is a context; that is some text containing a placeholder, a hole, such that if the placeholder is filled in with a proof of  $\Gamma \vdash T, \Delta$ , then the result is a proof of  $\Gamma \vdash \Delta$ . Furthermore, in this paper, the placeholder will, in the spirit of [1], always be at the very beginning of the rendering.

**command** an expression of the category  $c ::= \langle v \parallel e \rangle$ ; it combines a term and an environment (a provider and a consumer) typed by the same  $\Gamma, T$  and  $\Delta$  into a “closed” whole proving the sequent  $\Gamma \vdash \Delta$ , which is the type of  $c$ . The type of commands does not have a stoup (no singled out formula).

The natural language rendering of the command  $\langle v \parallel e \rangle$  thus naturally is the rendering of  $e$  with the placeholder filled in with the rendering of  $v$ . In this paper, this amounts to the concatenation of the rendering of  $v$  and the rendering of  $e$ .

## 2.2 Typing

In  $\bar{\lambda}\mu\tilde{\mu}$ , one makes use of a *hypothesis context* ( $\Gamma$ , which is a set of declarations  $\{x_1 : T_1, \dots, x_n : T_n\}$  where the  $T_i$  are simple types and all  $x_i$  are different) and a *goal context* ( $\Delta$ , which is a set of declarations  $\{\alpha_1 : T_1, \dots, \alpha_n : T_n\}$  where the  $T_i$  are simple types and all  $\alpha_i$  are different).

The syntactic categories of *terms*, *environments* and *commands* are

$$\begin{aligned} v &::= x \mid \lambda x : T. v \mid \mu \alpha : T. c \\ e &::= \alpha \mid v \circ e \mid \tilde{\mu} x : T. c \\ c &::= \langle v \parallel e \rangle \end{aligned}$$

The typing rules are:

$$\begin{array}{c} \Gamma, x : T \vdash x : T \mid \Delta \quad \Gamma \mid \alpha : T \vdash \alpha : T, \Delta \\ \frac{c : (\Gamma \vdash \alpha : T, \Delta)}{\Gamma \vdash (\mu \alpha : T. c) : T \mid \Delta} \quad \frac{c : (\Gamma, x : T \vdash \Delta)}{\Gamma \mid (\tilde{\mu} x : T. c) : T \vdash \Delta} \quad \frac{\Gamma \vdash v : T \mid \Delta \quad \Gamma \mid e : T \vdash \Delta}{\langle v \parallel e \rangle : (\Gamma \vdash \Delta)} \\ \frac{\Gamma \vdash v : T \mid \Delta \quad \Gamma \mid e : T' \vdash \Delta}{\Gamma \mid v \circ e : T \rightarrow T' \vdash \Delta} \quad \frac{\Gamma, x : T \vdash v : T' \mid \Delta}{\Gamma \vdash (\lambda x : T. v) : T \rightarrow T' \mid \Delta} \end{array}$$

**Definition 1.** A command whose environment ends in  $\alpha$  is said to conclude  $\alpha$ . It ultimately concludes  $\alpha$  if it concludes  $\alpha$  or ends in a binder (i.e.  $\tilde{\mu} x : T. c$ ) whose binding domain (i.e.  $c$ ) ultimately concludes  $\alpha$ .

For example,  $\langle v \parallel v' \circ \alpha \rangle$  concludes  $\alpha$ , but  $\langle v \parallel v' \circ \tilde{\mu} y : T. \langle v_0 \parallel v_1 \circ \alpha \rangle \rangle$  does not. The latter ultimately concludes  $\alpha$ .

**Intuitionistic Fragment** Intuitionistic logic is obtained by restricting the use of environment variables to only the most recently (innermost) bound one.

**Definition 2.**  $\alpha$  is said to be used intuitionistically in  $c$  iff it occurs only in positions where it is the most recently bound environment variable. Equivalently, no path leading to an occurrence of  $\alpha$  traverses a  $\mu$ .

## 2.3 Structure of $\bar{\lambda}\mu\tilde{\mu}$

**Reduction** Computationally,  $\mu$  binds the environment it is combined with; dually,  $\tilde{\mu}$  binds the term it is combined with, leading to these reduction rules, which form a critical pair:

$$\langle \mu \alpha : T. c \parallel e \rangle \rightsquigarrow c \{ \alpha := e \} \quad \langle v \parallel \tilde{\mu} x : T. c \rangle \rightsquigarrow c \{ x := v \}$$

In particular,  $\langle v \parallel \tilde{\mu} x : T. c \rangle$  is better known as `let  $x : T := v$  in  $c$` .  $\mu$  gives the current goal a name (so that it can be referred to later, e.g. switched back to) and gets the proof back in the “neutral” state where no goal is in focus. The direct equivalent of  $\beta$ -reduction in  $\lambda$ -calculus is

$$\langle \lambda x : T. v \parallel v' \circ e \rangle \rightsquigarrow \langle v' \parallel \tilde{\mu} x : T. \langle v \parallel e \rangle \rangle$$

The right hand-side then further reduces to  $\langle v \{ x := v' \} \parallel e \rangle$ .

## 2.4 Basic Pseudo-Natural Language Rendering

The purpose of this rendering is to be a purely depth-0 structural, left-to-right reading of  $\bar{\lambda}\mu\tilde{\mu}$  expressions, that is faithful to the proof the expression codes for (don't improve on it).  $\boxed{\leftarrow}$  is an increase in indentation level and  $\boxed{\leftarrow}$  a decrease.

$$\begin{array}{ll}
\llbracket x \rrbracket := \text{by } x & \llbracket \alpha \rrbracket := \boxed{\leftarrow} \text{ done proving } \alpha \\
\llbracket \lambda x : T.v \rrbracket := \text{assume } T(x) \llbracket v \rrbracket & \llbracket v \circ e \rrbracket := \text{and } \llbracket v \rrbracket \llbracket e \rrbracket \\
\llbracket \mu\alpha : T.c \rrbracket := \text{thesis } T(\alpha) & \llbracket \tilde{\mu}x : T.c \rrbracket := \text{we have proven } T(x) \\
\llbracket \boxed{\leftarrow} \llbracket c \rrbracket \rrbracket & \llbracket c \rrbracket \\
\llbracket \langle v \parallel e \rangle \rrbracket := \llbracket v \rrbracket \llbracket e \rrbracket & 
\end{array}$$

*Example 1.* This term

$$\begin{array}{l}
\lambda x_R : P \rightarrow R. \lambda x_P : Q \rightarrow S \rightarrow P. \lambda y_S : S. \lambda y_Q : Q. \mu\alpha : R. \\
\langle x_P \parallel y_Q \circ y_S \circ \tilde{\mu}y_P : P. \langle x_R \parallel y_P \circ \alpha \rangle \rangle
\end{array}$$

of type  $(P \rightarrow R) \rightarrow (Q \rightarrow S \rightarrow P) \rightarrow S \rightarrow Q \rightarrow R$  renders as

$$\begin{array}{ll}
\text{assume } P \rightarrow R & (x_R) \\
\text{assume } Q \rightarrow S \rightarrow P & (x_P) \\
\text{assume } S & (y_S) \\
\text{assume } Q & (y_Q) \\
\text{thesis } R & (\alpha) \\
\quad \text{by } x_P \text{ and by } y_Q \text{ and by } y_S & \\
\quad \text{we have proven } P & (y_P) \\
\quad \text{by } x_R \text{ and by } y_P & \\
\text{done proving } \alpha & 
\end{array}$$

The following term of the same type:

$$\begin{array}{l}
\lambda x_R : P \rightarrow R. \lambda x_P : Q \rightarrow S \rightarrow P. \lambda y_S : S. \lambda y_Q : Q. \mu\alpha : R. \\
\langle x_R \parallel (\mu\beta : P. \langle x_P \parallel y_Q \circ y_S \circ \beta \rangle) \circ \alpha \rangle
\end{array}$$

renders as

$$\begin{array}{ll}
\text{assume } P \rightarrow R & (x_R) \\
\text{assume } Q \rightarrow S \rightarrow P & (x_P) \\
\text{assume } S & (y_S) \\
\text{assume } Q & (y_Q) \\
\text{thesis } R & (\alpha) \\
\quad \text{by } x_R \text{ and thesis } P & (\beta) \\
\quad \text{by } x_P \text{ and by } y_Q \text{ and by } y_S & \\
\quad \text{done proving } \beta & \\
\text{done proving } \alpha & 
\end{array}$$

The previous term was a forward (bottom-up) proof, this is backward (top-down) proof. In this manner,  $\bar{\lambda}\mu\tilde{\mu}$  allows to choose at every step whether it is done backwards or forwards.

## 2.5 Enhanced Pseudo-Natural Language

We present several enhancements to the basic transformation, that do not break faithfulness to the proof the expression embodies. In order to keep the presentation simple, we will not discuss the interaction between the enhancements explicitly, unless there is an interesting or problematic point.

**Backwards Proofs** This enhancement, namely replacing “and thesis” by “the thesis is reduced to”, was already proposed in [1].

$$\begin{aligned} \llbracket (\mu\alpha : T.c) \circ e \rrbracket &:= \text{the thesis is reduced to } T & (\alpha) \\ &\quad \boxed{\hookrightarrow} \llbracket c \rrbracket \llbracket e \rrbracket \end{aligned}$$

It makes the intent of backwards proofs much more clear. Read the previous example again while mentally doing the replacement.

**Intuitionistic Logic** Here, we recognise when single-goal logic (i.e. intuitionistic logic augmented or not with a classical logic axiom) is used and adapt the rendering. This consists in omitting the “( $\alpha$ )” when rendering a  $\mu\alpha : T.c$  when  $\alpha$  is used intuitionistically in  $c$ , combined with these rules:

$$\begin{aligned} \llbracket \alpha \rrbracket &\text{ when the innermost parent } \mu \text{ binds } \alpha \\ &:= \boxed{\leftarrow} \text{ done} \\ \llbracket \mu\alpha : T.c \rrbracket &\text{ when } c \text{ ultimately concludes } \alpha \\ &:= \text{we have to prove } T \text{ } (\alpha) \end{aligned}$$

With this improvement, our natural language translation gives the same result as the one in [1] on single-goal (sub)proofs, and also handles multiple-goal proofs.

**Announcing Thesis Changes** The basic rendering has the feature that it informs the reader *only at the end* of a subproof that what has been proven was not what the reader thinks of as “the current thesis”. We see that e.g. in “done proving  $\beta$ ” or “we have proven  $T(x)$ ”. That is essentially inherited from a prefix depth-first left-right reading of  $\lambda\mu\tilde{\mu}$  terms. It enhances the readability of the proof if such changes are announced at the start of the corresponding subproof, rather than at the end. This is typically also required in the proof input language of proof assistants. There are essentially three situations where such a thesis change happens:

- Switching to another goal in  $\Delta$ , a thing that is implicit in the standard sequent calculus, but is made explicit by the stoup structure of  $\bar{\lambda}\mu\tilde{\mu}$ . This corresponds to the pattern  $\mu\alpha : T.c$  where  $c$  does not ultimately conclude  $\alpha$  (but, say  $\beta : T'$ ). We want to announce the thesis  $T'(\beta)$ , however, in a pattern like  $\mu\alpha : T.\langle v \parallel \tilde{\mu}x : T''.\langle v' \parallel c \rangle \rangle$  we want to delay the announcement until we are under the  $\tilde{\mu}$  binder. As a solution, we use a subscript in the

transformation to keep track of the thesis currently active in the natural language text.

$$\begin{aligned} \llbracket \langle v \parallel e \rangle \rrbracket_{\beta} & \text{ when } e \text{ does not conclude } \beta \text{ and concludes } \alpha \\ & := \text{we now consider thesis } \alpha \\ & \quad \llbracket v \rrbracket_{\alpha} \llbracket e \rrbracket_{\alpha} \\ \llbracket \langle \mu\beta : T.c \rangle \rrbracket_{\alpha} & := \text{thesis } T(\beta) \\ & \quad \boxed{\hookrightarrow} \llbracket c \rrbracket_{\beta} \end{aligned}$$

This rendering keeps implicit in the natural language text that the active thesis is the most recently introduced one.

- A cut. If the root of the term of a command is a  $\mu$ , then the basic rendering already makes the announcement; we just tweak the text a bit:

$$\begin{aligned} \llbracket \langle \mu\alpha : T.c \parallel e \rangle \rrbracket & := \text{we now prove } T(\alpha) \\ & \quad \boxed{\hookrightarrow} \llbracket c \rrbracket \\ & \quad \llbracket e \rrbracket \end{aligned}$$

As to the pattern  $\langle \lambda x : T.v \parallel e \rangle$ , it is dismissed as “bad style”: It is a proof that does a thesis change, but refuses to announce it. It is suggested to  $\eta$ -expand this term to  $\langle \mu\alpha : T.\langle v \parallel \alpha \rangle \parallel e \rangle$ , which can be done programmatically as part of a “proof enhancement” transformation.

- The pattern  $\langle v \parallel v_1 \circ \dots \circ v_n \circ \tilde{\mu}x : T.c \rangle$ . We can describe the environment part more succinctly by writing  $e(\tilde{\mu}x : T.c)$ : it is an environment that finishes with  $\tilde{\mu}x : T.c$  and  $e(\cdot)$  is that environment with the  $\tilde{\mu}$  removed and replaced by a placeholder  $\cdot$ . A rendering would be

$$\begin{aligned} \llbracket \cdot \rrbracket & := \boxed{\hookrightarrow} \text{ done} \\ \llbracket \langle v \parallel e(\tilde{\mu}x : T.c) \rangle \rrbracket & := \text{we now prove } T(x) \\ & \quad \boxed{\hookrightarrow} \llbracket v \rrbracket \llbracket e(\cdot) \rrbracket \\ & \quad \llbracket c \rrbracket \end{aligned}$$

Note that this rule and the “detect a cut” rule form a critical pair; we resolve it with

$$\begin{aligned} \llbracket \langle \mu\alpha : T.c \parallel \tilde{\mu}x : T.c' \rangle \rrbracket & := \text{we now prove } T(x, \alpha) \\ & \quad \boxed{\hookrightarrow} \llbracket c \rrbracket \\ & \quad \llbracket c' \rrbracket \\ \llbracket \langle \mu\alpha : T.c \parallel e(\tilde{\mu}x : T'.c') \rangle \rrbracket & := \text{we now prove } T'(x) \\ & \quad \boxed{\hookrightarrow} \text{we now prove } T(\alpha) \\ & \quad \quad \boxed{\hookrightarrow} \llbracket c \rrbracket \\ & \quad \quad \llbracket e(\cdot) \rrbracket \\ & \quad \llbracket c' \rrbracket \end{aligned}$$

These rules catch occurrences of  $\mu$  when its type is not the current thesis in the text; this allows to enhance the rendering of the other occurrences (those that capture the current thesis and give it a name):

$$\llbracket \mu\alpha : T.c \rrbracket := \text{left to prove: } T(\alpha) \\ \boxed{\hookrightarrow} \llbracket c \rrbracket$$

*Remark 1.* The rules introduced here have the big disadvantage that their structural depth (the depth at which they have to look into an expression before deciding how to render its root) is unbounded. This can be fixed by changing the syntax a bit, so that in a command the terminal constructors of the environment are available at depth 1:

$$\begin{aligned} E &::= \cdot | v \circ E && \text{all non-terminal environment constructors} \\ e &::= \alpha | \tilde{\mu}x : T.c && \text{all terminal environment constructors} \\ c &::= \langle v | E | e \rangle \end{aligned}$$

The meaning of the new syntax command  $\langle v | v_1 \circ \dots \circ v_n \circ \cdot | e \rangle$  is just  $\langle v | \llbracket v_1 \circ \dots \circ v_n \circ e \rrbracket \rangle$ . The typing rules can be adapted accordingly.

**From Binary to  $n$ -ary**  $\circ$  is a binary constructor, but one can recognise sequences of it and treat it as an  $n$ -ary constructor; this is here combined with controlling its interaction with term variables and  $\mu$  more closely:

$$\begin{aligned} \llbracket x_0 \circ x_1 \circ \dots \circ x_n \circ e \rrbracket &:= \text{by } x_0, x_1, \dots, x_{n-1} \text{ and } x_n \llbracket e \rrbracket \\ \llbracket \langle x | x_0 \circ x_1 \circ \dots \circ x_n \circ e \rangle \rrbracket &:= \text{by } x, x_0, x_1, \dots, x_{n-1} \text{ and } x_n \llbracket e \rrbracket \\ \llbracket (\mu\alpha_0 : T_0.c_0) \circ \dots \circ x_j \circ \dots \\ &\dots \circ (\mu\alpha_i : T_i.c_i) \circ \dots \circ e \rrbracket := \text{the thesis is reduced to:} \\ &\bullet T_0(\alpha_0) \\ &\quad \boxed{\hookrightarrow} \llbracket c_0 \rrbracket \\ &\dots \\ &\bullet \text{assumption } x_j \\ &\dots \\ &\bullet T_i(\alpha_i) \\ &\quad \boxed{\hookrightarrow} \llbracket c_i \rrbracket \\ &\dots \\ &\llbracket e \rrbracket \end{aligned}$$

Note that any environment of the shape  $v \circ e$ , where the root of  $v$  is a recursive constructor other than  $\mu$  (e.g.  $\lambda$ ) tends to give a rather unintelligible rendering; this is not fixed on purpose. The view is that such an expression encodes a proof in bad style; fixing it crosses the line of showing a *better* proof than the one written, not the proof written. It is best to handle this kind of improvements at the calculus level, by  $\eta$ -expanding such a  $v$  into  $\mu\alpha : T. \langle v | \alpha \rangle$ .

## 2.6 Other Constructors for Implication

There is a variety of alternative ways to handle implication in the calculus [6]. In our quest for a saturated calculus, we examine them all.

$\iota_2$  The first, and the only one we decide to keep as is, is called  $\iota_2$  in [6], but this conflicts with another constructor with the same notation; we thus rename it to  $\lambda_- : A$ , in line with the convention that  $_$  is a special name for “do not bind to a name”:

$$v ::= \dots | \lambda_- : A.v \qquad \frac{\Gamma \vdash v : T' | \Delta}{\Gamma \vdash (\lambda_- : T.v) : T \rightarrow T' | \Delta}$$

$\llbracket \lambda_- : T.v \rrbracket :=$  assumption  $T$  in thesis is not necessary.  $\llbracket v \rrbracket$

If the transformation has access to typing information (e.g. because every expression is annotated with its type), then one can use:

$\llbracket \lambda_- : T.v \rrbracket :=$  it suffices to prove  $t(v)$ .  $\llbracket v \rrbracket$

where  $t(v)$  is the type of  $v$ . This introduces some redundancy if a  $\mu$  follows immediately; this redundancy can be avoided with

$\llbracket \lambda_- : A.\mu\alpha : B.c \rrbracket :=$  it suffices to prove  $B$  ( $\alpha$ )  
 $\boxed{\Leftrightarrow} \llbracket c \rrbracket$

$\iota_1$  As to its companion  $\iota_1$ ,

$$v ::= \dots | \iota_1(e) \qquad \frac{\Gamma | e : A \vdash \Delta}{\Gamma \vdash \iota_1(e) : A \rightarrow B | \Delta}$$

while the logical step performed is naturally and immediately accepted as admissible, it is not intuitively seen as one atomic step; it is more natural to decompose it into two steps, namely assuming  $A$  and erasing goal  $B$ . We thus introduce a “no binding” version of  $\mu$ :

$$v ::= \dots | \mu_- : T.c \qquad \frac{c : (\Gamma \vdash \Delta)}{\Gamma \vdash (\mu_- : T.c) : T | \Delta}$$

$\llbracket \mu_- : T.c \rrbracket :=$  we give up on the current thesis  
 $\boxed{\Leftrightarrow} \llbracket c \rrbracket$

Furthermore, it is an instance of a bigger problem in the context of the rest of the natural language translation: term constructors that syntactically recurse into the environment category do not fit well. We have not found a nice phrase that turns what follows (which, being the translation of an environment, consumes a proof) into something which provides a proof. The best we could do was a rather weak and unnatural “the other goals follow from  $A \multimap \llbracket e \rrbracket$ ”, which had to

be combined with changing the translation for  $\tilde{\mu}x : T.c$  to “we can now assume  $T \perp \llbracket c \rrbracket$ ”, because e.g. in the expression  $\iota_1(\tilde{\mu}x : A.c)$  of type  $A \rightarrow B$ ,  $A$  has not been proven, but assumed, so “we have proven” does not fit anymore. It makes the translation of  $\tilde{\mu}$  in other situations weaker, but not wrong:

$$\langle \mu\alpha : T.\dots \parallel \tilde{\mu}x : T.\dots \rangle \quad \begin{array}{l} \text{thesis } T \quad (\alpha) \\ \dots \\ \text{done proving } \alpha \\ \text{we can now assume } T \quad (x) \\ \dots \end{array}$$

$\lambda(x : A, \alpha : B).c$  naturally feels like two steps and is advantageously replaced by  $\lambda x : A.\mu\alpha : B.c$ .

$$v ::= \dots | \lambda(x : A, \alpha : B).e \quad \frac{c : (\Gamma, x : A \vdash \alpha : B, \Delta)}{\Gamma \vdash (\lambda(x : A, \alpha : B).c) : A \rightarrow B | \Delta}$$

$\lambda\alpha : B$  takes an environment as argument, which raises the problems already discussed.

$$v ::= \dots | \lambda\alpha : B.e \quad \frac{\Gamma | e : T \vdash \alpha : B, \Delta}{\Gamma \vdash (\lambda\alpha : B.e) : A \rightarrow B | \Delta}$$

$\llbracket \lambda\alpha : B.e \rrbracket :=$  we now prove  $B$ , which follows from  $A$

$\mu_- : T.c$  has a natural dual in a putative  $\tilde{\mu}_- : T.c$ :

$$e ::= \dots | \tilde{\mu}_- : T.c \quad \frac{c : (\Gamma \vdash \Delta)}{\Gamma | (\tilde{\mu}_- : T.c) : T \vdash \Delta}$$

but while  $\mu_- : T.c$  fulfils a real role (e.g. in  $\lambda x : T_x.\mu_- : \perp. \langle x \parallel \beta \rangle$ , the current goal  $\perp$  really is not useful in the rest of the proof; it is thus not useful to bother to give it a fresh name  $\alpha$  to never refer to it later), any instance of  $\tilde{\mu}_- : T.c$  shows that the proof contains a completely non useful part: it takes the effort to prove  $T$ , but then just throws that result away.  $\langle x \parallel y \circ \tilde{\mu}_- : T. \dots \rangle$  would correspond to something like “by  $x$  and  $y$ , we have proven  $T$ , but don’t use that fact in the rest of the proof ...”.

## 2.7 Link to proof assistants

We here succinctly treat the transformation of intuitionistic-logic  $\bar{\lambda}\mu\tilde{\mu}$  terms to Isar proofs by way of an example. A transformation into Mizar is similar (except that Mizar can *only* do forward steps, no backward step). Also a transformation into PVS has been designed, but it is less direct. Both are not included here for lack of space. It should be noted that not all  $\bar{\lambda}\mu\tilde{\mu}$  terms can be mapped faithfully to an Isar proof: there are proof constructs in our saturated system that Isar cannot capture. We could syntactically single out the  $\bar{\lambda}\mu\tilde{\mu}$  terms that map to an Isar proof, but we take a different approach by describing a transformation of

$\bar{\lambda}\mu\tilde{\mu}$  terms. The terms that are invariant under this transformation are the ones corresponding to Isar proofs.

Replace any pattern in the left column by the one in the right column:

$$\begin{array}{ll}
\langle v \parallel \dots \circ (\lambda x : T.v') \circ \dots \rangle & \langle v \parallel \dots \circ (\mu\alpha : T_\alpha. \langle \lambda x : T.v' \parallel \alpha \rangle) \circ \dots \rangle \\
\langle \lambda x : T.v \parallel v_0 \circ \dots \rangle & \langle \lambda x : T.v \parallel \tilde{\mu}y : T'. \langle x \parallel v_0 \circ \dots \rangle \rangle \\
\langle v \parallel \dots \circ (\mu\alpha : T_\alpha.c) \circ x \circ \dots \rangle & \langle v \parallel \dots \circ (\mu\alpha : T_\alpha.c) \circ (\mu\beta : T_\beta. \langle x \parallel \beta \rangle) \circ \dots \rangle \\
\langle \mu\alpha : T_\alpha.c \parallel e \rangle & c\{\alpha := e\} \\
\lambda x : T.x' & \lambda x : T.\mu\alpha : T'. \langle x \parallel \alpha \rangle
\end{array}$$

Most of these transformation rules are interesting by themselves and fall under “the pattern on the left is bad proof style”, but others have their origins in idiosyncrasies of Isar. These rules identify the subcalculus of  $\bar{\lambda}\mu\tilde{\mu}$  that Isar proofs map to.

We do not deal here with mapping Isar proofs to  $\bar{\lambda}\mu\tilde{\mu}$ , but we claim that any Isar proof that does not use automation can be mapped faithfully to  $\bar{\lambda}\mu\tilde{\mu}$ . The Isar commands not used by the transformation below are mostly either syntactic sugar or logically equivalent to a basic command that we do use, or are concerned with automation.

The transformation follows. The purpose of `cl` is to count the lambdas in a term; we don’t spell out its definition. We use the alternate syntax of page 10 for clarity.

$$\begin{aligned}
\llbracket \mu\alpha : T_\alpha. \langle v \mid E \mid \tilde{\mu}x : T_x.c \rangle \rrbracket &:= \mathbf{have} \ x : T \ \llbracket \langle v \mid E \mid \alpha \rangle \rrbracket \\
&\quad \llbracket \mu\alpha : T_\alpha.c \rrbracket \\
\llbracket \mu\alpha : T.c \rrbracket &:= \mathbf{show} \ T \ \llbracket c \rrbracket \\
\llbracket \lambda x : T.v \rrbracket &:= \mathbf{assume} \ x : T \ \llbracket v \rrbracket \\
\llbracket \langle x \mid y_0 \circ \dots \circ y_{n-1} \circ \cdot \mid \alpha \rangle \rrbracket &:= \mathbf{by} \ (\mathbf{rule} \ \mathbf{mp}, \dots (n \ \text{times}) \dots, \ \mathbf{rule} \ \mathbf{mp}, \\
&\quad \mathbf{fact} \ x, \ \mathbf{fact} \ y_0, \ \dots, \ \mathbf{fact} \ y_{n-1})
\end{aligned}$$

$$\begin{aligned}
\llbracket \langle x \mid y_0 \circ \dots \circ y_{n-1} \circ v_0 \circ \dots \circ v_{p-1} \circ \cdot \mid \alpha \rangle \rrbracket &:= \\
&\mathbf{proof} \ (\mathbf{rule} \ \mathbf{mp}, \dots (n+p \ \text{times}) \dots, \ \mathbf{rule} \ \mathbf{mp}, \\
&\quad \mathbf{fact} \ x, \ \mathbf{fact} \ y_0, \ \dots, \ \mathbf{fact} \ y_{n-1}) \boxed{\longleftrightarrow} \\
&\quad \llbracket v_0 \rrbracket \\
&\quad \dots \\
&\quad \llbracket v_{p-1} \rrbracket \boxed{\longleftrightarrow} \\
&\mathbf{qed} \\
\llbracket \langle v \mid \cdot \mid \alpha \rangle \rrbracket &:= \\
&\mathbf{proof} \ (\mathbf{rule} \ \mathbf{impI}, \dots \mathbf{cl}(v) \ \text{times} \dots, \ \mathbf{rule} \ \mathbf{impI}) \boxed{\longleftrightarrow} \\
&\quad \llbracket v \rrbracket \boxed{\longleftrightarrow} \\
&\mathbf{qed}
\end{aligned}$$

In the rule for  $\llbracket \mu\alpha : T_\alpha. \langle v \mid E \mid \tilde{\mu}x : T_x.c \rangle \rrbracket$ ,  $\langle v \mid E \mid \alpha \rangle$  is not well-typed (the  $\alpha$  may be unbound or of the wrong type), but that doesn’t matter, because the ‘name’  $\alpha$  is never used in the Isar output (it translates to `qed`). One can vary on this

translation, producing different Isar proofs – that we claim have the same logical structure.

## 2.8 Classical Logic

We remind the reader that sequent calculus handles classical logic by maintaining a set of goals throughout the reasoning; concluding any one of these goals concludes the whole proof. In a  $\bar{\lambda}\mu\tilde{\mu}$  command, this set of goals is the set of all environment variables the command has a name for; that is  $\Delta$  in the typing judgement. For a  $\bar{\lambda}\mu\tilde{\mu}$  term, it is the environmental variables the term has a name for (the  $\Delta$  in the typing judgement), augmented with the type of the term (the type in the stoup), which does not have a name. A  $\mu\alpha : T.c$  is the binding of this unnamed goal  $T$  to the name  $\alpha$ , making it available at any point in  $c$ . As an example, we show a proof of Peirce’s law,  $((P \rightarrow Q) \rightarrow P) \rightarrow P$ :

$$\lambda x : (P \rightarrow Q) \rightarrow P. \mu\alpha : P. \langle x \parallel (\mu\beta : P \rightarrow Q. \langle \lambda y : P. \mu\gamma : Q. \langle y \parallel \alpha \parallel \beta \rangle) \circ \alpha \rangle$$

We show an alternative didactic rendering:

assume $(P \rightarrow Q) \rightarrow P$	(x)
left to prove: $P$	(α)
we now consider thesis $P$ (α)	
by $x$ the thesis is reduced to $P \rightarrow Q$ (or $P$ (α))	(β)
we now consider thesis $P \rightarrow Q$ (β)	
assume $P$	(y)
left to prove: $Q$ (or any of $P$ (α), $P \rightarrow Q$ (β))	(γ)
we now consider thesis $P$ (α)	
by $y$	

The classical logic step is the second “we now consider thesis  $\alpha$ ”. Intuitionistically, one would have to prove  $Q$  at this point, but we announce “no, we are switching to goal  $\alpha$ ”, conclude it, which concludes the whole proof.

In a saturated calculus that covers complete propositional logic, there should be constructs to handle classical logic the usual way, that is with a double negation law and excluded middle, De Morgan laws, classical decomposition of disjunction, etc.

## 3 Future Work

There are several directions in which this can be taken further. The most obvious is extending, with the same concern for saturation, to some predicate logic. The second glaring need is that the problem of capturing automation in theorem provers needs to be addressed for the language to be really functional in practise as a proof interchange language. A natural idea is to store as part of the term a witness provided by the automation, which would be used to produce a proof in a system whose automation is weaker. Alas, it may not be practical or possible to get a witness from some provers’ automation.

The proof of the pudding being in the eating, we will concretely implement the transformations from and to various proof languages (various proof assistants, but also e.g. a standard sequent calculus); this would find a natural place as part of a proof assistant.

On a more theoretical side, our natural language rendering has an underlying concept of structure of a natural language proof, which (when restricted to single-goal logic) is quite close to the structure of declarative proof languages like Mizar or Isar. But, in particular in its notion of active thesis, and when a change of it needs to be explicitly announced (i.e. always), it is not in complete agreement with the structure of proofs in  $LK_{\mu\tilde{\mu}}$  (sequent calculus with ‘stoup’, basically just  $\bar{\lambda}\mu\tilde{\mu}$  without term/expression/command information). One would like changes in the active thesis to be characterised as either a cut, or an explicit active thesis change. We will thus define a sequent calculus whose notion of cut matches the notion of introducing an arbitrary new thesis (a forward step) in our natural language, and that matches the natural language’s need for an explicit switch action between the theses. Restricting the hypothesis-creating children of a left introduction rule to a left introduction rule or an axiom may be the main thing needed to achieve the former. It would then be interesting to study proof intent conserving transformations between that calculus,  $LK_{\mu\tilde{\mu}}$ , standard stoup-free sequent calculus and other proof formats.

## References

1. Sacerdoti Coen, C.: Explanation in natural language of  $\bar{\lambda}\mu\tilde{\mu}$  terms. In Kohlhase, M., ed.: MKM2005. Volume 3863 of LNAI., Springer (2006) 234–249
2. Sacerdoti Coen, C.: Declarative representation of proof terms. In: Prog. Lang. for Mechanized Math. Volume 07-10 of RISC Reports., University of Linz (2007) 3–18
3. Kirchner, F.: Interoperable proof systems. PhD thesis, École Polytechnique (2007)
4. Autexier, S., Sacerdoti Coen, C.: A formal correspondence between OMDoc with alternative proofs and the  $\bar{\lambda}\mu\tilde{\mu}$ -calculus. In Borwein, J.M., Farmer, W.M., eds.: MKM2006. Volume 4108 of LNAI., Springer (2006) 67–81
5. Curien, P.L., Herbelin, H.: The duality of computation. In: ICFP ’00. SIGPLAN Notices 35(9), ACM (2000) 233–243
6. Herbelin, H.: C’est maintenant qu’on calcule; au cœur de la dualité. Habilitation à diriger des recherches, Université Paris 11 (december 2005)
7. Asperti, A., Guidi, F., Padovani, L., Sacerdoti Coen, C., Schena, I.: Mathematical knowledge management in HELM. AMAI **38(1)** (may 2003) 27–46
8. Mamane, L.E., Geuvers, H., McKinna, J.: A logically saturated extension of  $\bar{\lambda}\mu\tilde{\mu}$  to propositional logic [http://www.mamane.lu/science/lbmmt\\_extension/](http://www.mamane.lu/science/lbmmt_extension/).
9. Corbineau, P.: A declarative proof language for the Coq proof assistant. In: TYPES 2007. Volume 4941 of LNCS., Springer (2007)
10. Wenzel, M.: Isar - a generic interpretative approach to readable formal proof documents. In Bertot et al, Y., ed.: TPHOLs’99. Volume 1690 of LNCS., Springer
11. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In Kapur, D., ed.: CADE-11. Volume 607 of LNAI., Springer (jun 1992) 748–752
12. Trybulec, A.: Mizar language <http://mizar.org/language/>.
13. Coscoy, Y.: Explication textuelles de preuves pour le calcul des constructions inductives. PhD thesis, Université de Nice-Sophia-Antipolis (September 2000)

## A Basic Propositional Logic

$v ::= \dots |(v, v)|\iota_{1,2}(v)|[\alpha : A, \beta : B].c|\lambda_{1,2}\alpha : A.v|\lambda_{\neg}x : A.v|\times|\text{TE}(T)|\text{DN}(v)$

$e ::= \dots |\pi_{1,2}[e]|(x : A, y : B).c|\lambda_{1,2}x : A.e|[e, e']|\neg[v]|\times|\text{DN}(e)$

$$\begin{array}{c}
\frac{\Gamma \vdash v : T|\Delta \quad \Gamma \vdash v' : T'|\Delta}{\Gamma \vdash (v, v') : T \wedge T'|\Delta} \quad \frac{c : (\Gamma, x : T, x' : T' \vdash \Delta)}{\Gamma|((x : T, x' : T').c) : T \wedge T' \vdash \Delta} \\
\frac{\Gamma, x : T|e : T' \vdash \Delta}{\Gamma|(\lambda_1 x : T.e) : T \wedge T' \vdash \Delta} \quad \frac{\Gamma, x : T'|e : T \vdash \Delta}{\Gamma|(\lambda_2 x : T'.e) : T \wedge T' \vdash \Delta} \\
\frac{\Gamma|e : T \vdash \Delta}{\Gamma|\pi_1[e] : T \wedge T' \vdash \Delta} \quad \frac{\Gamma|e : T' \vdash \Delta}{\Gamma|\pi_2[e] : T \wedge T' \vdash \Delta} \\
\frac{\Gamma \vdash v : T|\Delta}{\Gamma \vdash \iota_1(v) : T \vee T'|\Delta} \quad \frac{\Gamma \vdash v : T'|\Delta}{\Gamma \vdash \iota_2(v) : T \vee T'|\Delta} \quad \frac{\Gamma, x : T \vdash v : \perp|\Delta}{\Gamma \vdash (\lambda_{\neg}x : T.v) : \neg T|\Delta} \\
\frac{c : (\Gamma \vdash \beta : T', \alpha : T, \Delta)}{\Gamma \vdash ([\alpha : T, \beta : T'].c) : T \vee T'|\Delta} \quad \frac{\Gamma|e : T \vdash \Delta \quad \Gamma|e' : T' \vdash \Delta}{\Gamma|[e, e'] : T \vee T' \vdash \Delta} \\
\frac{\Gamma \vdash v : T'|\alpha : T, \Delta}{\Gamma \vdash (\lambda_1 \alpha : T.v) : T \vee T'|\Delta} \quad \frac{\Gamma \vdash v : T|\alpha : T', \Delta}{\Gamma \vdash (\lambda_2 \alpha : T'.v) : T \vee T'|\Delta} \\
\frac{}{\Gamma \vdash \times : \top|\Delta} \quad \frac{\Gamma \vdash v : T|\Delta}{\Gamma|\neg[v] : \neg T \vdash \Delta} \quad \frac{}{\Gamma|\times : \perp \vdash \Delta} \\
\frac{}{\Gamma \vdash \text{TE}(P) : P \vee \neg P|\Delta} \quad \frac{\Gamma \vdash v : \neg \neg T|\Delta}{\Gamma \vdash \text{DN}(v) : T|\Delta} \quad \frac{\Gamma|e : T \vdash \Delta}{\Gamma|\text{DN}(e) : \neg \neg T \vdash \Delta}
\end{array}$$

$[\iota_1(v)] :=$  it suffices to prove the left part of the disjunction  $[v]$

$[\iota_2(v)] :=$  it suffices to prove the right part of the disjunction  $[v]$

$[\lambda_{1,2}\alpha : A.v] :=$  keeping in mind that we may prove  $A$  ( $\alpha$ ),  $[v]$

$[(x : A, y : B).c] :=$  we have proven  $A$  ( $x$ ) and  $B$  ( $y$ )

$[c]$

$[\lambda_{1,2}x : T.e] :=$  we have proven  $T$  ( $x$ ) and  $[e]$

$[\text{DN}(v)] :=$  proof by contradiction

$[\text{DN}(e)] :=$  and by double negation elimination

$[(v, v')] := \bullet [v]$

$\bullet [v']$

$[[\alpha : A, \beta : B].c] :=$  thesis  $A$  ( $\alpha$ ) or  $B$  ( $\beta$ )

$\boxed{\leftrightarrow} [c]$

$[[e, e']] :=$  either

$\boxed{\leftrightarrow} [e]$

or

$\boxed{\leftrightarrow} [e'] \boxed{\leftrightarrow}$

$[\pi_{1,2}[e]] :=$  in particular  $[e]$

$[\neg[v]] :=$  and  $[v]$

$\boxed{\leftrightarrow}$ done (ECQ)

$[\times] :=$  true

$[\times] := \boxed{\leftrightarrow}$ done (EFQ)

$[\lambda_{\neg}x : A.v] :=$  assume  $A$  ( $x$ )  $[v]$

$[\text{TE}(T)] :=$  by TE

ECQ is “ex contradictione (sequitur) quodlibet”, EFQ “ex falso (sequitur) quodlibet” and TE “(principium) tertii exclusi”.

This introduces constructs that should be treated like a  $\mu$ , respectively like a  $\tilde{\mu}$  or like a  $\alpha$  by the enhanced rendering. There is also not anymore unicity of the terminal environment constructor e.g. in  $[\alpha, \tilde{\mu}x : T.c]$ . The adaptations of the enhanced rendering to this are not detailed here. Some of these constructs are inherently not intuitionistic; this needs to be taken into account when restricting oneself to intuitionistic logic.