Explanation in Natural Language of $\bar{\lambda}\mu\tilde{\mu}$-terms

Claudio Sacerdoti Coen*

Project PCRI, CNRS, École Polytechnique, INRIA, Université Paris-Sud.
sacerdot@cs.unibo.it

Abstract. The $\bar{\lambda}\mu\tilde{\mu}$-calculus, introduced by Curien and Herbelin, is a calculus isomorphic to (a variant of) the classical sequent calculus LK of Gentzen. As a proof format it has very remarkable properties that we plan to study in future works. In this paper we embed it with a rendering semantics that provides explanations in pseudo-natural language of its proof terms, in the spirit of the work of Yann Coscoy [3] for the λ-calculus. The rendering semantics unveils the richness of the calculus that allows to preserve several proof structures that are identified when encoded in the λ-calculus.

1 Introduction

An important topic of Mathematical Knowledge Management (MKM) is the definition of standards for the representation of mathematical documents at different semantical levels (presentation, content, semantics using the terminology of [1]). The current situation for mathematical expressions is almost satisfactory: MathML Presentation is a W3C standard for the presentation level, and the lack of MathML rendering engines has been solved; OpenMath is a de facto standard for the content level, and several tools already integrate phrasebooks for communicating formulae in OpenMath according to a given content dictionary; the interactive theorem proving community is slowly starting to consider open formats for replacing the proprietary semantic encodings or just for communication with external tools. On the contrary, there is no mature format for proofs at the content level. The only candidate is the OMDoc standard, that integrates a module for proofs since its first version. However, the original format was not expressive enough for describing in a natural way the proofs of the HELM[1] library. Thus the proof module was redesigned almost from scratch in the MoWGLI[2] European Project, and the new proposal will be part of the forthcoming OMDoc 1.2 standard [7]. A rendering semantics (i.e. a default explanation of the proofs in a pseudo-natural language) is also provided by MoWGLI [2]. However, a serious third party evaluation of the new proposal has not been done and there exists no test suite of proofs that can be used to assess the flexibility of the format.

* Partially supported by 'MoWGLI: Math on the Web, Get it by Logic and Interfaces', EU IST-2001-33562

[1] http://helm.cs.unibo.it

[2] http://mowgli.cs.unibo.it

To try to improve the situation the first step consists in fixing a few requirements that a proof format for the content level must satisfy. Here is our list:

1. **Flexibility.** It must be possible to encode both rigorous, human provided, proofs and proofs that are generated from their semantics level. The encoding should respect the *structure* of the proof, avoiding the identification of proofs that differ in their structure. What the structure of a proof is is already a non trivial question. For instance, proof nets or natural deduction identify more proofs than sequent calculus. For presentational purposes we are interested in identifying as few proofs as possible, up to their structure only. For instance, a top down proof should not be identified with its bottom up counterpart. However, a content encoding must identify proofs that have the same structure and that differ only up to rhetorical text.
2. **Annotations.** It must be possible to decorate the proof structure with rhetorical text. The rhetorical text is the presentational counterpart of the proof content. It is requested only for consumption by humans.
3. **Explanation in Natural Language.** The format must have a *rendering semantics* associated to it. That is, it must be possible to generate rhetorical text that describes the proof structure. The generated text is not required to be nice to read or close to the text that a mathematician would choose. Annotations are explicitly provided to deal with the situation where a nice presentational proof is required. The rendering semantics is useful, for instance, when the proof is automatically generated from a semantics proof — say, created using a proof assistant by mimicking a pen&paper proof — and the user needs to check whether the pen&paper proof that she wants to formalize and the proof generated by the proof assistant are actually the same proof.
4. **A Clear Semantics.** This is surely the most controversial point. On the one hand we are talking about a content level format, that should not be restricted to the proof steps that are correct in just one foundation and one logic; on the contrary it should capture the usual rigorous but informal style of the proofs of real world mathematicians. On the other hand it must describe a proof, and not a document with an arbitrary structure; it must allow for simple checks, as for references to hypotheses out of scope or for the well nesting of subproofs; it must allow for proof transformations, such as cut elimination. In other words, it must be as close as possible to a calculus without becoming a semantic encoding instead of a content level encoding.

The OMDoc 1.2 proof module strives to achieve the points 1–3. However, its semantics is someway defined a posteriori and it is not fully understood nor made explicit.

Via the Curry-Howard isomorphism, several λ-calculi can be seen as proof formats at the semantic level for the logics they are isomorphic to. As proof formats they can be equipped with a rendering semantics [3] and extending them with annotations is also a trivial exercise. However, they lack flexibility. A partial reason is that, being at the semantics level, they are bound to a precise

logic. However, there are deeper reasons that are illustrated in Sect. 2 and that are not related to their focus on a particular logic. Thus they are not a good model to build a content level proof format on.

In a seminal paper in 2000 [5] Curien and Herbelin proposed the $\bar{\lambda}\mu\tilde{\mu}$-calculus that is isomorphic to (a variant of) the classical sequent calculus LK of Gentzen. I claim that this calculus is a perfect proof format at the semantics level and that it is inherently very flexible. To obtain a content level calculus from it it is just necessary to relax a bit its interpretation by decoupling it from its logic. Moreover, I also claim that it has several remarkable similarities with OMDoc 1.2 and in a future work I plan to make this relation explicit by providing a bisimulation of OMDoc into the $\bar{\lambda}\mu\tilde{\mu}$-calculus that respects the rendering semantics. As a preliminary step in that direction, in this paper I will provide a rendering semantics to the $\bar{\lambda}\mu\tilde{\mu}$-calculus that is extremely intuitive and unveils all the good features of the calculus as a proof format.

2 A $\bar{\lambda}\mu\tilde{\mu}$-calculus Primer.

The $\bar{\lambda}\mu\tilde{\mu}$-calculus [5] is an extremely elegant synthesis of the $\bar{\lambda}$-calculus of Herbelin [6] and the $\lambda\mu$-calculus of Parigot [9]. The $\bar{\lambda}$-calculus of Herbelin is a λ-calculus that is isomorphic to (a variant of) the *intuitionistic sequent calculus* LJ of Gentzen. The $\lambda\mu$-calculus of Parigot is a λ-calculus that is isomorphic to *classical, multi conclusions, natural deduction*. The $\bar{\lambda}\mu\tilde{\mu}$-calculus is isomorphic to (a variant of) the *classical sequent calculus* LK of Gentzen. However, the interest of the calculus is that it is not a simple merge of two existing calculi; on the contrary, it is greatly superior to both of them since it makes explicit for the first time at the syntactic level two fundamental dualities of the computation:

1. Terms vs Contexts
2. Call-by-name vs Call-by-value

We will explain the two dualities in detail. Before that, however, we notice that this result is, a posteriori, not very surprising. Indeed the classical sequent calculus is well known for its meta-theoretical properties, since it reveals the deep symmetries of the logical connectives that are hidden in natural deduction and since it can also be seen as a fine grained analysis of natural deduction, especially for cut elimination. Thus it is natural that a λ-calculus isomorphic to LK should be the best framework for the study of the symmetries of computation. What is not absolutely obvious, however, is that these two dualities are deeply connected with the flexibility of the proof format. Let's explain this.

Terms vs Contexts A context is an expression with exactly one placeholder $\Box$ for a "missing' term. The placeholder can be filled with a term to obtain a placeholder-free expression. The placeholder can be typed with the type of the expected term, and only terms of the expected type can fill the placeholder. A context can apply its placeholder to arguments ($\Box\ \bar{t}$) or it can bind a name to it (**let** $x := \Box$ **in** c) to refer to it later on, for instance to pass it to a function.

Dually, a term can be seen as an expression with exactly one placeholder]_[for a "missing context" that is "all around" the term. The placeholder can be filled with a context to obtain a placeholder-free expression. The placeholder can be typed with the type of the term, and only contexts that expects a term of the expected type can fill the placeholder. A term can wait for inputs from its context $(]\lambda\overline{x}.t[)$ or — in languages with control operators like Scheme's CALL/CC — it can bind a name to it $(]\mu\alpha.c[$, μ is the binder and α the bound name) to refer to it later on.

Now consider an expression without placeholders and imagine it to be isomorphic to a proof of some thesis from some set of hypotheses. The expression can be broken to be seen as the composition of a term and a context whose placeholders are given "the same type" T (actually, a dual type; we will be more precise later). The term and the context can be thought respectively as "a proof of T from the hypothesis" and "a proof of the thesis from T". Thus in the term the type of the placeholder represents what must be proved as a first step in the proof, and the placeholder is the rest of the proof. In the context the type of the placeholder represents what was proved so far and the placeholder is the proof so far. The operators that are used to bind the placeholder in a term and in a context can be thought as ways of stating or manipulating the (local) conclusion(s) (for a term), or as ways of stating or manipulating the (local) hypotheses (for a context). This kind of manipulation is very frequent in pen&paper proofs, where an intermediate result can be claimed (binding a context), a label can be associated to intermediate results for further reference (binding a term), a proof of an intermediate result can be postponed (a context that binds its term is displayed before the term), or the current thesis can be reduced to another one by anticipating the rest of the proof (a term that binds a context is displayed after the context).

Call-by-name vs Call-by-value What are the dynamics of call-by-value and call-by-name? The first strategy processes the arguments before processing the function; the second strategy processes the function until it needs to process the arguments. If you substitute "explains" or "prove" with "process" you will obtain the definition of the bottom-up and top-down proof styles. A bottom-up proof proves (process the argument) a result (the type of the argument) before using it later on (processing the function). A top-down proof prooves the thesis (process the function) until it has reduced the thesis to an easier one (the type of the argument) that is then proved (the argument is processed).

Usually, call-by-name and call-by-value are global strategies that are applied in the reduction of a functional program (a λ-expression). In the $\bar{\lambda}\mu\tilde{\mu}$-calculus, instead, there exists at the syntactic level both call-by-value related and call-by-name related redexes (and a third form of redexes whose strategy is not yet fixed and that can non-deterministically reduce towards one of the other two redexes, but this is not important in our discussion). Thus the $\bar{\lambda}\mu\tilde{\mu}$-calculus is flexible enough to distinguish between top-down and bottom-up proof steps, while this is not possible in the plain λ-calculus (unless we play tricks, as using β-expansion

to "mark" bottom-up steps or we extend the calculus with a **let** ... **in** construct that is native of the $\bar{\lambda}\mu\tilde{\mu}$-calculus).

Since we think that the intuition we just provided is someway deeper than the gory technical details we are shortly going to present, we prefer to reinforce it by explaining it again along a different axis. As we already said, the $\bar{\lambda}\mu\tilde{\mu}$-calculus is a beautiful synthesis of the $\bar{\lambda}$-calculus and the $\lambda\mu$-calculus, made completely symmetric by adding a $\tilde{\mu}$ operator (the **let** ... **in** in a more usual syntax). We give now the intuition about what is the contribution for flexibility (as a proof format) of each component.

$\bar{\lambda}$ The $\bar{\lambda}$-calculus establishes a Curry-Howard isomorphism with a sequent calculus. A sequent calculus identifies far fewer proofs than natural deduction, which is Curry-Howard isomorphic to the λ-calculus. In particular, top-down and bottom-up proofs are distinguished in a sequent calculus derivation (where it is recorded if the user eliminates a rule on the left hand side first or on the right hand side first). In natural deduction, instead, top-down vs bottom-up corresponds to the order of construction of the derivation (from the leafs to the root or from the root to the leaves), but both procedures at the end produce exactly the same tree (unless cuts are artificially introduced to mark the bottom-up steps). This is one reason why the sequent calculus provides a more fine-grained analysis of the process of construction of the derivation and, in our context, it gives more flexibility in proof representation.

$\tilde{\mu}$ The **let** $x : T := \Box$ **in** c (that we will soon write $\tilde{\mu}x : T.c$ to show the beautiful symmetries of the calculus) gives a label (x) to the last result proved ($\Box$) and it makes explicit its type T. The label is used later on to refer to the result. The type makes explicit what is the conclusion of the last proof step (the "last" proof step of $\Box$).

This construct is necessary for a proof format since it allows to reuse a subproof more than once, without replicating a proof, and since it is used to associate to a subproof its thesis. In the λ-calculus a redex can be used for sharing a proof, partially simulating the $\tilde{\mu}$. Moreover, since the semantics of $\tilde{\mu}$ is that of a bottom-up proof (since it gives a label to the previous proof step), redexes can be rendered as bottom-up proofs. Notice that in a typed calculus the binder in a redex also associates to the proof (the argument of the application in the redex) its thesis (its type). If we manage to avoid the redex trick, however, we have to guess the type that is no longer recorded by the binder. If the type system is decidable, the type can be automatically inferred. However, since we do not expect applications that adopt a proof format to integrate a type inference engine and since we want to impose no semantics (no choice of any type system) to our proof format, we need to pre-compute the type of the argument and explicitly store it in the proof format. Actually, we need to store the type of each subterm (we call this an inner-type in [1]).

The need for inner-types is evident when we recall that a λ-term is isomorphic to a derivation in natural language in the sense that you can obtain the λ-term from the derivation by erasing from the tree all the conclusions of the rule (i.e. what a user would keep in a pen&paper proof) since they can be inferred from the rules themselves (i.e. what a user throws away in a pen&paper proof) and the tree structure. Thus in our proof format we are obliged to reconstruct from the λ-term every inner-type, we need to keep the structure of the term, but we can throw away the term! (When the actual terms, i.e. the justifications of each proof step, are thrown away, we obtained a *proof sketch* in the terminology proposed by Wiedijk [10]).

Thus the $\bar{\lambda}\mu\tilde{\mu}$-calculus is superior to the λ-calculus since recording of the inner-types and the $\tilde{\mu}$ (or **let** ... **in**) is already part of the calculus, while it needs to be introduced in the λ-calculus.

Just to be precise, notice also that the $\tilde{\mu}$ construct can be simulated in the λ-calculus as a redex only from the point of view of the reduction. On the contrary, the typing rule for $\tilde{\mu}$ is not equivalent since in **let** $x : T := t$ **in** c we can type c under the assumption that x is equal to t, which is stronger than the assumption x has type T (for instance, when the type system admits dependent types).

μ The control operator μ that binds the context of a term to reuse it later has a surprising role. It is introduced in the calculus to capture classical logic and, when the calculus is seen as a proof format, it is used to give a label and to state explicitly what is the thesis that is going to be proved next. The relation with classical logic is obvious: when multiple μ are in scope the expression has visibility of several possible conclusions at once, and it can dynamically choose to conclude any one of them. This clearly corresponds to a sequent with several conclusions.

However, a pen&paper proof, even a classical one, never uses multiple conclusions. Indeed, natural deduction with several conclusions (the logic the $\lambda\mu$-calculus of Parigot is Curry-Howard isomorphic to) is not natural at all, as the classical sequent calculus is not. Most mathematicians prefer to work in an intuitionistic natural deduction setting augmented with one or more equivalent classical axiom such as excluded middle or double negation elimination.

Thus we can easily argue that a proof format is not requested to support proofs with multiple conclusions, if not for completeness reasons. Thus we can argue that we will not need the ability of the μ constructor of associating a label to the thesis we want to prove next. Indeed in pen&paper proofs a thesis is never labelled. However a proof format does need a way to state what the user is going to prove next, since this construct is often used by mathematicians to clarify the proof or to postpone parts of it. Once again, this construct is already native in the $\bar{\lambda}\mu\tilde{\mu}$-calculus, and in the λ-calculus it can only be simulated with a redex. Notice, however, that too many different things must already be simulated with a redex in the λ-calculus. In other words, once again we realise that the λ-calculus is not expressive enough to be a reasonable proof format.

Hoping to have transmitted all of our intuitions to the reader, we are now ready to briefly dive into the details of the calculus. The syntax is described first. The reduction and typing rules can be found in the appendixes. For the metatheory and the proof of its remarkable properties the reader can consult the literature, starting from [5] where the calculus has been defined.

2.1 Syntax

The $\bar{\lambda}\mu\tilde{\mu}$-calculus has three syntactic categories: *terms* (that include term variables $x, y, z, \ldots$); *environments* — or contexts — (that include context or continuation variables $\alpha, \beta, \gamma, \ldots$); and *commands* obtained by replacing the placeholder of an environment with a term (or, dually, by replacing the placeholder of a term with an environment, as already explained).

For each syntactic category we give both the $\bar{\lambda}\mu\tilde{\mu}$-calculus and the usual syntax in λ-calculus notation.

		$\bar{\lambda}\mu\tilde{\mu}$-syntax	usual syntax
Term	$v ::=$	x	x
		$\mid \lambda x : T.v$	$\lambda x : T.v$
		$\mid \mu\alpha : T.c$	
Environment	$E ::=$	α	
		$\mid v \circ E$	$E[(\Box\ v)]$
		$\mid \tilde{\mu} x : T.c$	$\mathbf{let}\ x : T := \Box\ \mathbf{in}\ c$
Command	$c ::=$	$\langle v \| \| E \rangle$	$E[v]$

The term variable x is bound by λ in v and by $\tilde{\mu}$ in c; the environment variable α is bound by μ in c. Notice the (syntactic for now) duality between μ and $\tilde{\mu}$. The only two constructors that have no syntactic dual are λ and $\circ$ (pronounced "cons"). In [5] the calculus is made perfectly symmetric by adding duals for λ and $\circ$. This extended version of the calculus is Curry-Howard isomorphic with classical subtractive sequent calculus [4]. We do not consider the subtractive case now, but we will comment on that in Sect. 4.

The "intuitionistic" fragment of the calculus, i.e. the fragment that is Curry-Howard isomorphic to the intuitionistic sequent calculus, is obtained by a simple syntactic restriction: only one environment variable is allowed (we denote it by $\star$ instead of using a Greek letter to make explicit that it is unique). Since only one variable is available, every μ constructor will override $\star$, so that only the latter continuation is in scope. This corresponds to the fact that the intuitionistic sequent calculus is obtained by restricting the sequents to have just one conclusion.

For the sake of completeness we give the reduction and typing rules of the calculus in App. A and B. They are taken without modification from [5]. The typing and reduction rules will not play any major role in the rest of the paper. However, we will exploit the possibility of inferring a type for each $\bar{\lambda}\mu\tilde{\mu}$-expression (by means of the typing rules) and of recording it directly in the term (by means

of a μ or $\tilde{\mu}$-expansion rule, see App. A). In the λ-calculus it is also possible to infer the type of a subexpression, but the type cannot be recorded in the term without introducing explicit type assignment operators.

3 Structural natural language rendering

We are now ready to provide (pseudo-)natural language rendering rules for the $\bar{\lambda}\mu\tilde{\mu}$-calculus. Before that we present similar rules for the λ-calculus, inspired by [3].

In both cases we attempt a *structural* translation, i.e. we try to associate to a term t its pseudo-natural language rendering $[\![t]\!]$ by structural recursion over t. We will also struggle to perform recursion over the direct subterms of t only and we will avoid processing the result of the recursive calls. Forcing the usual terminology, we will call *structural* a translation that respects all these properties.

Ideally, we would also require the translation to preserve the order of the subterms: if A and B are two sibling subterms in the proof and if A precedes B, than the rendering of A must precede that of B. This additional constraint — that surprisingly is satisfied for the $\bar{\lambda}\mu\tilde{\mu}$-calculus — makes extremely easy for a human being to "invert the transformation", building by hand the term from its rendering.

Our interest in a structural translation derives from our interest in the properties of the calculus as a proof format. For sure with complex, non-structural translations we can improve the generated text, aiming at more natural sentences. However, we claim that a good proof format must have a simple rendering semantics: if generating natural language for the proofs encoded in the proof format requires major proof transformations we consider this a serious fault of the proof format. Moreover, especially when we are interested in generating explanations of formal proofs proved with an interactive or automatic theorem prover, we do require the rendering semantics to be simple and structural to avoid loosing confidence on the correctness of the proof we are examining.

For technological reasons, every proof format should be equipped with an XML concrete syntax, imposing XSLT as the standard language for describing transformations on the document. Notice that the expressive power of XSLT (1.0) is, in practice, extremely close to our second definition of structural transformation. Indeed XSLT does not allow to process the result of a recursive call (a Result Tree Fragment) and only simple recursive functions can be described in a concise way[3].

[3] XSLT is a Turing complete purely functional language. However, Turing completeness derives from the fact that a Result Tree Fragment (a tree) can be converted to a string for further processing and that every data type (e.g. the state of a Turing machine) can be encoded in a string and manipulated with general recursion. In practice, however, working with strings is quite cumbersome in an ad-hoc language designed to transform trees.

3.1 λ-calculus

$[\![x]\!] := \text{consider } x$

$[\![\lambda x : T.t]\!] := \text{suppose } T \text{ (x)}$
$[\![t]\!]$

$[\![(\dots(t\ t_1)\ \dots\ t_n)]\!] :=$ $[\![t_1]\!]$
we proved T_1 (H_1)
$\dots$
$[\![t_n]\!]$
we proved T_n (H_n)
$[\![t]\!]$
we proved T (H)
by H, H_1, $\dots$, H_n

At first we observe that the transformation is not really structural since for the case of application we process the inner term t_1 before the outer term t_n. Notice that in the $\bar{\lambda}\mu\tilde{\mu}$-calculus the application $(\dots(t\ t_1)\dots t_n)$ is turned inside out, becoming $\langle t||t_1 \circ (\dots \circ (t_{n-1} \circ t_n)\dots)\rangle$ and making the transformation structural!

We can now repeat several of the observations we already made when discussing the intuitions about the $\bar{\lambda}\mu\tilde{\mu}$-calculus. In every rule one or more inner-types T are lacking and type inference is required to reconstruct them beforehand. Since there is just one construct, application, to derive new facts, bottom-up and top-down proofs are identified. Thus we need to represent all the proofs in the same way. The rule we provided renders every proof step in a bottom-up way, processing the t_i before using them to conclude T'. In this case, not only inner-types T_i are missing for the t_i, but also fresh labels H_i. A structural rule to render applications in a mixed bottom-up/top-down way can be easily provided:

$[\![(t\ t_1)]\!] := [\![t]\!]$
we proved T (H)
by H we reduce the thesis to T_1
$[\![t_1]\!]$

where T and T_1 are the inner types of t and t_1.

However, the latter rule does not solve the lack of flexibility that derives from having to choose a uniform style of rendering every applications (bottom-up vs top-down). Notice also that the latter rule introduces a mixed proof style since t is rendered as a bottom-up step. This cannot be avoided unless an ad-hoc rule is provided for redexes.

The solution that provides more flexibility by forcing a particular interpretation of redexes can be obtained adding the rule

$[\![(\lambda x : T.t\ t_1)]\!] := [\![t_1]\!]$
we proved T (x)
$[\![t]\!]$

and by replacing the rule for application with

$[\![(x\ t_1)]\!] :=$ by x we reduce the thesis to T_1
$[\![t_1]\!]$

Notice that in this way we impose a normal form on the λ-terms: every application $(t\ t_1)$ where t is not a variable must be β-expanded to $(\lambda x : T.(x\ t_1)\ t)$, that is semantically equivalent (according to our rendering semantics) to the mixed top-down bottom-up rule for application given before. We will not spend

more time on improvements for the rendering semantics of the λ-calculus, since in the $\bar{\lambda}\mu\tilde{\mu}$-calculus these problems simply disappear.

We conclude by showing as a small example the λ-term that corresponds to a proof of $A \Rightarrow (A \Rightarrow B) \Rightarrow (B \Rightarrow C) \Rightarrow C$ and its structural natural language rendering. We use superscripts to record in the λ-term the inner-type and label of a sub-term.

	suppose A (H), suppose $A \Rightarrow B$ (AB)
	suppose $B \Rightarrow C$ (BC)
$\lambda H : A.\lambda AB : A \to B.$	consider H; we proved A (H')
$\lambda BC : B \to C.$	consider AB; we proved $A \Rightarrow B$ (AB')
$(BC\ (AB^{AB':A\Rightarrow B}\ H^{H':A})^{K:B})^{:C}$	by AB', H' we proved B (K)
	consider BC; we proved $B \Rightarrow C$ (BC')
	by BC', K

Notice again that, due to lack of structurality, it is difficult to transform the λ-term to its textual counterpart looking at the λ-term only. Building the λ-term from the text — a plausible operation if we consider the calculus a proof format — is even more complex. Indeed, the natural language really corresponds to the equivalent (up to β-expansions) λ-term
$\lambda H : A.\lambda AB : A \to B.\lambda BC : B \to C.(\lambda H' : A.(\lambda AB' : A \to B.(\lambda K : B.(\lambda BC' : B \to C.(BC'\ K)\ BC)\ (AB'\ H'))\ AB)\ H)$ that is simpler for a human to render in natural language, but still quite annoying since the eyes must wonder back and forth between the λ-abstractions and their arguments in redexes. Only introducing **let** ... **in** and replacing redexes with them it becomes possible to read the term in natural language (and to produce the term by hand from the natural language!) without any major effort.

3.2 $\bar{\lambda}\mu\tilde{\mu}$-calculus

We provide now a similar but completely structural rendering semantics for the $\bar{\lambda}\mu\tilde{\mu}$-calculus. According to the intuitions we provided, we should associate one or more sentences to a term, a textual context (i.e. a text with a placeholder) to an environment and we should render a command by filling the placeholder of its environment with the text obtained by its term. However, we anticipate that our semantics is so well behaved that the placeholder (that we will leave implicit) is always at the beginning of the text. Thus rendering a command simply amounts to concatenating the two generated texts.

$$[\![\langle v||E\rangle]\!] := [\![v]\!]\ [\![E]\!]$$

$[\![x]\!]$	:= by x	$[\![\alpha]\!]$	:= $\boxed{\hookleftarrow}$ done
$[\![\lambda x : T.t]\!]$	:= suppose T (x)	$[\![t \circ E]\!]$	:= and $[\![t]\!]$
	$[\![t]\!]$		$[\![E]\!]$
$[\![\mu\alpha : T.c]\!]$	:= we need to prove T	$[\![\tilde{\mu}x : T.c]\!]$	:= we proved T (x)
	$\boxed{\hookrightarrow}$ $[\![c]\!]$		$[\![c]\!]$

The symbols $\boxed{\hookrightarrow}$ and $\boxed{\hookleftarrow}$ stand for the increase/decrease of the indentation.

We provide as an example two $\bar{\lambda}\mu\tilde{\mu}$-terms that correspond to two different proofs of $A \Rightarrow (A \Rightarrow B) \Rightarrow (B \Rightarrow C) \Rightarrow C$.

Fully bottom-up proof:

$\mu\star : A \to (A \to B) \to (B \to C) \to C$	we need to prove $A \Rightarrow (A \Rightarrow B) \Rightarrow (B \Rightarrow C) \Rightarrow C$
$\langle \lambda H : A . \lambda AB : A \to B .$	suppose A (H); suppose $A \Rightarrow B$ (AB)
$\lambda BC : B \to C .$	suppose $B \Rightarrow C$ (BC)
$\mu\star : C .$	we need to prove C
$\langle AB \vert\vert H \circ \tilde{\mu} K : B .$	by AB and by H we proved B (K)
$\langle BC \vert\vert K \circ$	by BC and by K
$\star \rangle \rangle$	done
$\vert\vert \star \rangle$	done

Fully top-down proof:

$\mu\star : A \to (A \to B) \to (B \to C) \to C$	we need to prove $A \Rightarrow (A \Rightarrow B) \Rightarrow (B \Rightarrow C) \Rightarrow C$
$\langle \lambda H : A . \lambda AB : A \to B .$	suppose A (H); suppose $A \Rightarrow B$ (AB)
$\lambda BC : B \to C .$	suppose $B \Rightarrow C$ (BC)
$\mu\star : C .$	we need to prove C
$\langle BC \vert\vert$	by BC
$\mu\star : B .$	and we need to prove B
$\langle AB \vert\vert$	by AB
$\mu\star : A .$	and we need to prove A
$\langle H \vert\vert$	by H
$\vert\vert \star \rangle$	done
$\vert\vert \star \rangle$	done
$\vert\vert \star \rangle$	done
$\vert\vert \star \rangle$	done

As made obvious by the two examples, all the rendering rules are not only structural, but they also preserve the order of the subterms. Thus it is very easy to read a $\bar{\lambda}\mu\tilde{\mu}$-term from left to right mentally producing the corresponding natural language. Dually, it is very easy to translate a proof sketch or a pen&paper proof to a $\bar{\lambda}\mu\tilde{\mu}$-term, a fundamental property for a proof format.

Notice also that indentation directives are "already present" in the term: indentation must be incremented when a μ is found and it must be decremented when a $\star$ is met. Moreover, an indented sub-proof can easily be hidden to the user by an interactive interface, showing only its thesis. The HELM library[4] adopts this strategy to increase usability by giving to the user a partial form of control over the level of details. The user can simply click on a hidden proof to unfold it, requesting more details.

In Sect. 3.1 we did not consider indentation directives. However, indentation rules cannot be avoided in the transformation to make explicit the scope of the hypotheses. Indeed the user can be deceived by the too simplified rendering semantics proposed for the λ-calculus. This does not happen for the $\bar{\lambda}\mu\tilde{\mu}$-calculus.

While the proof of the first example seems very readable, that of the second example is not. However, if you replace "and we need to prove" with the more appealing (and equally semantically faithful) sentence "we reduce the thesis to"

[4] http://helm.cs.unibo.it

you will get a totally reasonable text: "... we need to prove C; by BC we reduce the thesis to B; by AB we reduce the thesis to A; by H done ...". This and other similar improvements can be implemented by trading off a little bit the property of the transformation being structural.

4 Generalization and improvements

The structural rendering semantics provided in the previous section confirms our intuitions about the fact that the $\bar{\lambda}\mu\tilde{\mu}$-calculus is a natural candidate for being a good proof format. Indeed it satisfies properties 1 (flexibility), 3 (explanation in natural language) and 4 (a clear semantics) given in the introduction. Property 2 (annotations) can be easily obtained by associating rhetorical text to each constructor of an expression. Since expressions are rendered in a structural way from left to right, associating placeholders in the rhetorical text to subexpressions is often as simple as matching the i-th placeholder with the i-th direct subexpression (instead of permutating the direct subexpressions or picking a subexpression that is deeper in the term).

As Yann Coscoy did for the λ-calculus [3] and as it has been done in a more incisive way in the MoWGLI project, we can trade the naturality of the generated text with the complexity of the rendering semantics. Since we are already starting from a much more structural and simple semantics and since the language is much richer, we can hope to obtain better results.

As we did for the $\bar{\lambda}\mu\tilde{\mu}$-calculus, it is also possible to get rid of expressions that have a weird explanation in natural language by imposing a normal form on the terms. An example is the renaming $\tilde{\mu}$ redex: $\langle H||\tilde{\mu}K : T.c\rangle$ ("by H we proved T (K); $[\![c]\!]$") can be reduced to $c\{K/H\}$. More generally, redexes correspond to cuts and cuts are detours in the proof. Cut elimination (i.e. reduction) can be applied to get rid of the unwanted detours. Due to lack of space we omit the analysis of the weird redexes associated to the semantics we provide and of the associated normal form that solves the problem. We only remark that to reach the normal form it is sufficient to either reduce the redexes or η-like-expand one subexpression of the redex according to the reduction rules of the calculus.

Of course, a calculus that is Curry-Howard isomorphic to the implicative fragment of the propositional calculus is not very interesting. The $\bar{\lambda}\mu\tilde{\mu}$-calculus can be easily extended to be in correspondence with stronger logics. In particular, we modified Fellowship[5], an experimental sequent calculus based proof assistant for first order logic developed by Florent Kirchner, to produce extended $\bar{\lambda}\mu\tilde{\mu}$-calculus proof terms. We have also already defined and implemented the structural rendering semantics for the extended calculus and we plan to enhance the generated text in the near future. The extension to the constructors that are related to the other connectives (negation, conjunction, disjunction and first order universal and existential quantification) have reserved no surprises and have not broken any good property of the calculus.

[5] http://www.lix.polytechnique.fr/Labo/Florent.Kirchner/fellowship

According to our initial claim, we can exploit the $\bar{\lambda}\mu\tilde{\mu}$-calculus in two different ways. Either as a format for proof terms in a proof assistant or as a general proof format. In the first case we should ask whether the non-standard sequent calculus the calculus is isomorphic too is reasonable to develop proofs in. Our short experience with Fellowship shows that as a sequent calculus it is indeed very interesting and pleasant to work with, especially for automation purposes. Indeed the distinguished formula acts as the linear hypothesis/continuation that must be eliminated next, reducing the search space. As a result Fellowship exposes only three tactics, axiom, cut and elim. The latter does not need as an argument the formula that must be eliminated, since it always act on the current distinguished (or focused) formula. More on this subject can be found in the literature about the calculus.

With respect to natural deduction, we remark that sequent calculus is always clumsier to work with interactively. We can easily adapt a natural deduction based system to produce $\bar{\lambda}\mu\tilde{\mu}$-calculus proof terms, since the sequent calculus is more fine grained than natural deduction. However, according to our initial remarks, we need to do it very carefully to obtain proof terms that record all the details of the process of construction of the proof, without identifying, for instance, top down and bottom up proofs. Concretely doing it by adapting an existent proof assistant based on natural deduction is other future work we plan to start.

We already remarked that classical proofs are usually presented in an intuitionistic logic extended with classical axioms, and not by handling multiple conclusions at once. Thus, to render classical proofs in Fellowship, we implemented a simple translation from classical $\bar{\lambda}\mu\tilde{\mu}$-expressions (i.e. expressions were there are occurrences of continuation variables that are not bound from the innermost enclosing μ binder) to intuitionistic $\bar{\lambda}\mu\tilde{\mu}$-expressions. Of course, to do so we need to introduce in the calculus a family of distinguished constants $\mathbf{EM}_T$ that inhabits the excluded middle for each type T. The translation can be easily implemented by structural recursion over the $\bar{\lambda}\mu\tilde{\mu}$-expressions:

$\mathcal{F}_\sigma(\mu\alpha : T.c) := \mu\alpha : T.\mathcal{F}_\sigma(c)$ if α is used intuitionistically
$\mathcal{F}_\sigma(\mu\alpha : T.c) := \mu\alpha : T.\langle \mathbf{EM}_T || \lambda H : T.H \circ \lambda H : \neg T.\mathcal{F}_{(\alpha,T,H)::\sigma}(c)$ otherwise
$\mathcal{F}_\sigma(\alpha) := \tilde{\mu}x : \neg T.\langle H || x \circ \xi\rangle$ if $(\alpha, T, H) \in \sigma$
$\mathcal{F}_\sigma(\alpha) := \alpha$ otherwise

All the other cases call $\mathcal{F}_\sigma$ recursively over each subexpression. The distinguished constant $\mathbf{EM}_T$ has type $(T \to T') \to (\neg T \to T') \to T'$ for each type T', and ξ ("ex falso sequitur quodlibet") is a distinguished continuation of type $\bot$ (i.e. a continuation that expects a term of type $\bot$ to conclude the proof). When the calculus is extended with disjunction, $\mathbf{EM}_T$ can be typed as $T \vee \neg T$ and $\mathcal{F}_\sigma$ must be slightly modified to use the $\bar{\lambda}\mu\tilde{\mu}$-calculus constructor that corresponds to case analysis (elimination of $\vee$ on the left hand side of a sequent).

Notice that the translation is purely syntactical and it does not depend on the typing judgement or on the reduction rules.

The translation is particularly effective, allowing to unveil the mathematical intuition that underlies a proof developed in a multi-conclusion sequent calculus (and that is usually extremely complex to grasp looking at the derivation only).

However, the translation often introduces lots of redexes that complicate the proof. Automatic elimination of weird redexes is probably mandatory as a post-processing step to obtain natural proofs.

5 Conclusions and perspectives

We have found yet another remarkable property of the $\bar{\lambda}\mu\tilde{\mu}$-calculus: it admits a very simple and structural rendering semantics, i.e. a translation from expressions (that are Curry-Howard isomorphic to proofs) to pseudo-natural language text. The calculus is so rich that it is able to differentiate between bottom-up and top-down proof steps, and it permits to label each intermediate result, also stating its thesis. Translating by hand a pen&paper proof sketch into a $\bar{\lambda}\mu\tilde{\mu}$-expression preserving the natural language (up to the rhetorical text) is also quite simple. Annotations can be added later on to the term to retrieve the original language.

Our impression is that, as a proof format, the $\bar{\lambda}\mu\tilde{\mu}$-calculus is as flexible as OMDoc. We plan to make this statement more precise in a forthcoming paper by providing a mutual translation between OMDoc and the $\bar{\lambda}\mu\tilde{\mu}$-expressions that respects the rendering semantics provided to both calculi in this work and in the MoWGLI project (for OMDoc).

We have also extended Fellowship, a proof assistant prototype developed by Florent Kirchner for first order logic, to record proofs as $\bar{\lambda}\mu\tilde{\mu}$-expressions, and we have equipped it with natural language rendering of the proofs. The rendering semantics implemented, being almost the one described in this paper, already produces readable explanations, but we plan to improve them in the near future by introducing new normal forms for $\bar{\lambda}\mu\tilde{\mu}$-expressions that avoid unnatural proof constructions. We have also implemented a translator of $\bar{\lambda}\mu\tilde{\mu}$-expressions to Coq proof terms and we are implementing a similar translator to Mizar and Isabelle/ISAR scripts. Automatic generation of OMDoc documents is also planned. In all these cases the aim is not only that of showing that a translation is possible, but also understanding the relative expressivity of these languages as proof formats by trying to preserve the rendering semantics of the $\bar{\lambda}\mu\tilde{\mu}$-expressions. Fellowship can be downloaded from

http://www.lix.polytechnique.fr/Labo/Florent.Kirchner/fellowship

References

1. A. Asperti, F. Guidi, L. Padovani, C. Sacerdoti Coen and I. Schena. "Mathematical Knowledge Management in HELM". In Annals of Mathematics and Artificial Intelligence, 38(1): 27–46, May 2003.
2. A. Asperti, C. Sacerdoti Coen. "Stylesheets to intermediate representation" (prototypes D2.c-D2.d) and I. Loeb, "Presentation stylesheets" (prototypes D2.e-D2.f), technical reports of MoWGLI (project IST-2001-33562).
3. Y. Coscoy. *Explication textuelles de preuves pour le calcul des constructions inductives*. PhD. thesis, Université de Nice-Sophia-Antipolis, 2000.

4. T. Crolard. "Subtractive logic". In Theoretical computer science, 254:1–2(2001), 151–185.
5. P. Curien, H. Herbelin. "The duality of computation". In Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00), ACM, SIGPLAN Notices 35(9), ISBN:1-58113-2-2-6, 233–243, 2000.
6. H. Herbelin. *Séquents qu'on calcule: de l'interprétation du calcul des séquents comme calcul de lambda-terms et comme calcul de stratégies gagnantes.* PhD. thesis, 1995.
7. M. Kohlhase. *OMDoc: An Open Markup Format for Mathematical Documents (Version 1.2).*
8. S. Lengrand. "Call-by-value, call-by-name, and strong normalization for the classical sequent calculus". In B. Gramlich and S. Lucas editors, Electronic Notes in Theoretical Computer Science, 86(4), Elsevier, 2003.
9. M. Parigot. "$\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction". In Proc. of the International Conference on Logic Programming and Automated Reasoning (LPAR), LNCS 624.
10. F. Wiedijk. "Formal Proof Sketches". In S. Berardi, M. Coppo and F. Damiani eds., Types for Proofs and Programs: Third International Workshop, TYPES 2003, LNCS 3085, 378–393, 2004.

A $\bar{\lambda}\mu\tilde{\mu}$-calculus Reduction Rules

We present the reduction rules both in $\bar{\lambda}\mu\tilde{\mu}$-calculus syntax and in the usual λ-calculus syntax, omitting the contextual rules to propagate reduction everywhere in an expression. As usual, the reduction rules correspond to cut elimination.

$\bar{\lambda}\mu\tilde{\mu}$-syntax		usual syntax	
$\langle \mu\alpha : T.c \| \| E \rangle$	$\triangleright\ c\{E/\alpha\}$	$E[\mu\alpha : T.c]$	$\triangleright\ c\{E/\alpha\}$
$\langle v \| \| \tilde{\mu}x : T.c \rangle$	$\triangleright\ c\{v/x\}$	**let** $x : T := v$ **in** c	$\triangleright\ c\{v/x\}$
$\langle \lambda x : T.v_1 \| \| v_2 \circ E \rangle$	$\triangleright\ \langle v_2 \| \| \tilde{\mu}x : T.v_1 \circ E \rangle$	$E[(\lambda x : T.v_1\ v_2)]$	$\triangleright\ E[$**let** $x : T := v_2$ **in** $v_1]$

We report just a few standard observations on the calculus that can be found and are explained in [5]. First of all notice that the μ and $\tilde{\mu}$ reduction rules are perfectly dual, whereas the rule for λ is asymmetric. Its dual rule is present in the subtractive system. Secondly, notice that the μ and $\tilde{\mu}$ rules form a critical pair. Giving priority to the μ rule imposes a call-by-value strategy to the calculus; the dual priority leads to call-by-name. Finally, observe that any redex is a command, but that there are commands that are not redexes. There exists a variant of the calculus where every command is a redex [8]. We have not investigated yet the property of these as proof formats.

The rules we have just presented are similar (and related) to β-reduction rules in the λ-calculus. The $\bar{\lambda}\mu\tilde{\mu}$-calculus can also be extended with rules that correspond to η-expansion. These rules are important for us since we can use them to put expressions in a normal form before rendering them in pseudo-natural language.

$$\begin{array}{ll} \mu\text{-expansion:} & v \Rightarrow \mu\alpha : T.\langle v \| \| \alpha \rangle \\ \tilde{\mu}\text{-expansion:} & E \Rightarrow \tilde{\mu}x : T.\langle x \| \| E \rangle \end{array}$$

$$(\textsc{cut})\quad \frac{\Gamma \vdash v : T | \Delta \qquad \Gamma | E : T \vdash \Delta}{\langle v || E \rangle : (\Gamma \vdash \Delta)}$$

$$(\textsc{Ax-r})\quad \frac{}{\Gamma; x : T \vdash x : T | \Delta} \qquad \frac{}{\Gamma | \alpha : T \vdash \alpha : T; \Delta}\quad (\textsc{Ax-l})$$

$$(\textsc{Impl-r})\quad \frac{\Gamma; x : T \vdash v : T' | \Delta}{\Gamma \vdash \lambda x : T.v : T \to T' | \Delta} \qquad \frac{\Gamma \vdash v : T | \Delta \qquad \Gamma | E : T' \vdash \Delta}{\Gamma | v \circ E : T \to T' \vdash \Delta}\quad (\textsc{Impl-l})$$

$$(\tilde{\mu})\quad \frac{c : (\Gamma \vdash \alpha : T; \Delta)}{\Gamma \vdash \mu\alpha : T.c : T | \Delta} \qquad \frac{c : (\Gamma; x : T \vdash \Delta)}{\Gamma | \tilde{\mu} x : T.c : T \vdash \Delta}\quad (\mu)$$

Table 1. Typing rules

In the previous two rules T is the type of v (respectively of E). Type inference is required in the general case to compute T. However, for each term v (or environment E) we can always precompute its type once and for all, recording it explicitly in the expression by means of a μ-expansion (a $\tilde{\mu}$-expansion). This property is exploited when the calculus is used as a proof format.

B $\bar{\lambda}\mu\tilde{\mu}$-calculus Typing Rules

A typing judgement is associated to each syntactic category of the calculus:
$\Gamma \vdash v : T | \Delta$, $\Gamma | E : T \vdash \Delta$, $c : (\Gamma \vdash \Delta)$

In all three kind of judgements the context Γ is a list of assumptions (i.e. a list of typed term variables $x_i : T_i$) and Δ is a list of continuations (i.e. a list of typed context variables $\alpha_i : T_i$). Notice that types associated to terms are differentiated from types associated to environments (i.e. the type expected for the term that will fill the placeholder). The former are written on the left hand side of the turnstile, whereas the latter are written on the right hand side.

A command is typed with the sequent $\Gamma \vdash \Delta$ that associates a type to every free variable in the command. Terms and environments are typed with sequents that associate types to every free variable and that are "enriched" with a distinguished formula, on the right hand side for terms and on the left hand side for environments. The distinguished formula is the type of the term or, dually, the type of the placeholder.

The Curry-Howard correspondence with classical sequent calculus should be evident from the typing rules given in Table 1 (where the distinguished formula can be considered at first just as a normal formula).

Observe that the symmetries of the calculus are perfectly respected at the typing level. For instance a term is given type $A \to B$ (on the right hand side of the sequent) when it waits for an input of type A to provide an output of type B. Dually an environment is given type $A \to B$ (on the left hand side of the sequent) when it provides an input of type A and waits for an output of type B.