A Curry-Howard-De Bruijn Isomorphism Modulo

Benjamin Wack

LORIA & Université Henri Poincaré, Nancy, France
E-mail: `Benjamin.Wack@loria.fr`

Abstract

The rewriting calculus combines in a unified setting the frameworks and capabilities of rewriting and λ-calculus. Its most general typed version, called Pure Pattern Type Systems (P^2TS) and adapted from Barendregt's λ-cube, is especially interesting from a logical point of view.

We show how to use a subset of P^2TS as a proof-term language for natural deduction modulo, extending the Curry-Howard-De Bruijn isomorphism for this class of logical formalisms. The pattern matching featured in the calculus allows us to model any congruence given by a term rewriting system.

We characterize how proofs can be denoted by P^2TS terms and we discuss the interest of our proof-term language for the issue of cut elimination. Finally, we explore some relations between our proof-term language and other formalisms: extraction of λ-terms and/or rewrite rules from P^2TS-terms, but also automated generation of proof-terms by a rewriting-based language.

1 Introduction

From the early beginnings of proof assistants based on type theory [7], it has been noticed that a high computational power would be needed in order to develop purposeful proofs. The evolution of the Coq proof assistant [13] is emblematic: its base formalism, the Calculus of Constructions introduced by Coquand and Huet [5], combines the deductive flexibility of higher-order logic and dependent types with the expressive power of polymorphism. It was later extended with primitive inductive types by Coquand and Paulin-Mohring [6] to get an enhanced convertibility relation on types. Latest developments explore the introduction of rewriting into the conversion relation [4], giving more and more computational power to the system.

While all these works aim at increasing the automation of an actual proof assistant, from a theoretical point of view, the *Poincaré principle* [1] states that computation steps can occur in a provable proposition without changing the proof of this proposition. A way to formalize this principle is deduction modulo, proposed by Dowek, Hardin and Kirchner [8], where the inference rules of a proof system (natural deduction for instance) are all defined modulo a congruence on propositions. This congruence is usually given by a term rewriting system.

In this work, we propose to relax the Poincaré principle, in the sense that we design proof-terms for deduction modulo keeping track of the computation steps. The task of proving can still be computer-assisted, so it does not get more difficult. Furthermore, we obtain expressive proof witnesses that enable us to strengthen the *de Bruijn criterion* [1]: the proof-checking kernel can be smaller, hence easier to verify by hand. Such enriched proof-terms are also likely to help understanding the interaction between computation and deduction.

The base language we use for proof-terms is an instance of the Pure Pattern Type Systems (P^2TS) [2]. These typed versions of the rewriting calculus[1] embed the proof-theoretic expressiveness of the λ-calculus together with versatile pattern matching mechanisms. Strong normalization of the simply-typed version was proved in [14], promoting this type theory as a good candidate for a consistent logical system featuring rewriting.

The main contributions of this work are:

- a proof of strong normalization for P^2TS with dependent types;
- an enhanced matching algorithm in P^2TS allowing for product types as patterns and dealing efficiently with α-conversion;
- an extended Curry-Howard isomorphism between proofs in deduction modulo and P^2TS terms;

The paper is organized as follows. In Sections 2 and 3, we recall the main definitions and properties respectively

[1]An extensive bibliography on the rewriting calculus can be found at `http://rho.loria.fr`

of deduction modulo and P^2TS, and we prove strong normalization for P^2TS with dependent types. Section 4 details the representation of proofs in deduction modulo with P^2TS-terms. In Sections 5 and 6, we explain the need and the applications for two technical aspects of our proof-terms, namely right-to-left use of rewrite rules and pattern matching modulo α-conversion. Finally, in Section 7 we formalize the isomorphism between terms and proofs and we discuss the issue of cut elimination; in Section 8 we see how we can relate our proof-terms with other formalisms.

2 Deduction modulo

Deduction modulo was introduced by Dowek, Hardin and Kirchner as a way to remove computational arguments from proofs by reasoning modulo a congruence on propositions [8]. In this paper, we will focus on a version of natural deduction modulo for intuitionistic first-order logic, so that we can use the λ-calculus with dependent types embedded into P^2TS.

2.1 Intuitionistic natural deduction modulo

Different presentations can be found for deduction modulo. Generally, one takes the usual deduction rules and adds a side condition so that either the conclusion or a premise can be converted.

Since our goal is to keep a precise account of computation steps, throughout this paper, we will consider an equivalent presentation using the usual deduction rules (without side conditions) and an explicit conversion rule ($\cong$). The implicational fragment of intuitionistic first-order logic is recalled in Fig. 1; the system can be extended with rules for introduction and elimination of disjunction, conjunction, existential quantification and falsehood.

All along this paper, the symbol $\cong$ will denote the congruence we consider in deduction modulo.

Let us see two examples showing the use of deduction modulo. In the following, the plain equality symbol $=$ is a predicate of arity 2.

Example 1 (A proof using conversion on terms) *When dealing with group theory, the properties of the neutral element e are generally given as a bunch of propositions like $\forall x.(e{*}x = x)$. However, we may consider that this is more related to computing the value of an expression, and reflect it in a congruence: $e * x \cong x$. With this setting, one can prove that e is the only neutral element without using*

$$\frac{}{\Gamma \vdash_{\cong} \varphi}\,(Ax) \qquad \text{if } \varphi \in \Gamma$$

$$\frac{\Gamma, \varphi \vdash_{\cong} \psi}{\Gamma \vdash_{\cong} \varphi \Rightarrow \psi}\,(\Rightarrow I)$$

$$\frac{\Gamma \vdash_{\cong} \varphi \Rightarrow \psi \qquad \Gamma \vdash_{\cong} \varphi}{\Gamma \vdash_{\cong} \psi}\,(\Rightarrow E)$$

$$\frac{\Gamma \vdash_{\cong} \varphi}{\Gamma \vdash_{\cong} \forall x.\varphi}\,(\forall I) \qquad \text{if } x \notin \mathcal{FV}(\Gamma)$$

$$\frac{\Gamma \vdash_{\cong} \forall x.\varphi}{\Gamma \vdash_{\cong} [t/x]\varphi}\,(\forall E)$$

$$\frac{\Gamma \vdash_{\cong} \varphi}{\Gamma \vdash_{\cong} \psi}\,(\cong) \qquad \text{if } \varphi \cong \psi$$

Figure 1. Natural deduction modulo

Leibniz equality axioms:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\forall y.(y{*}e' = y) \vdash_{\cong} \forall y.(y{*}e' = y)}\,(Ax)}{\forall y.(y{*}e' = y) \vdash_{\cong} e{*}e' = e}\,(\forall E)}{\forall y.(y{*}e' = y) \vdash_{\cong} e' = e}\,(\cong)\ \textit{with } e{*}e' \cong e'}{\vdash_{\cong} \forall y.(y{*}e' = y) \Rightarrow e' = e}\,(\Rightarrow I)$$

Example 2 (A proof using conversion on propositions) *Supposing we want to formalize $\mathbb{Z}$ with three constructors 0, p and s (resp. for predecessor and successor), we will use a congruence on terms:*

$$s(p(z)) \cong z \cong p(s(z))$$

but we may also define equality with a congruence on propositions:

$$s(x){=}y \ \cong\ x{=}p(y) \quad \textit{and} \quad p(x){=}y \ \cong\ x{=}s(y)$$

Those definitions allow us to prove the statement:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{s(a){=}s(b) \vdash_{\cong} s(a){=}s(b)}\,(Ax)}{s(a){=}s(b) \vdash_{\cong} a{=}p(s(b))}\,(\cong)\ \textit{with } s(a){=}s(b) \cong a{=}p(s(b))}{s(a){=}s(b) \vdash_{\cong} a{=}b}\,(\cong)\ \textit{with } p(s(b)) \cong b}{\vdash_{\cong} s(a){=}s(b) \Rightarrow a{=}b}\,(\Rightarrow I)$$

The congruence $\cong$ is generally defined via a rewriting system ($\mathcal{R}$), the left-hand sides being either terms or atomic propositions. The congruence $\cong$ is then defined as the reflexive, transitive, symmetric and contextual closure of the rewriting relation $\mapsto_{\mathcal{R}}$. The two previous examples can be presented respectively by the rewriting systems

$$e * x \to x$$

and

$$\begin{cases} s(p(z)) \to z \\ p(s(z)) \to z \\ s(x) = y \to x = p(y) \\ p(x) = y \to x = s(y) \end{cases}$$

Both of the considered congruences are decidable. For the first one, since the rewriting system is confluent and terminating, the convertibility of two terms can be decided by simply comparing their normal forms. The second one is not confluent, making convertibility less immediate to check (but possible, by completion of the rewriting system).

2.2 Proof-terms and cut elimination

To our knowledge, proof terms for deduction modulo were studied only by Dowek and Werner in [9]. Their work is strongly aimed at proving cut elimination. Therefore, they do not use the rule ($\cong$), but have convertibility side conditions on every rule, *e.g.* introduction of implication becomes:

$$\frac{\Gamma, \varphi \vdash_{\cong} \psi}{\Gamma \vdash_{\cong} \chi} \ (\Rightarrow I) \qquad \text{if } \chi \cong \varphi \Rightarrow \psi$$

As a consequence, their proof terms only keep track of the deduction steps: congruence is always implicit, and typechecking depends on a decision procedure for convertibility of two propositions.

Definition 1 (λ-proof-terms) *The implicative fragment is characterized by the typing rules from Fig. 2.*

Example 3 *The inferences from examples 1 and 2 have respective λ-proof-terms $\lambda\alpha.(\alpha e)$ and $\lambda\alpha.\alpha$.*

These proof-terms are suitable *witnesses*, in the sense that they contain enough information for proof reconstruction whenever the congruence $\cong$ is decidable. Moreover, they can be used for proving cut elimination for a broad class of congruences. However, they are too abstract to really *represent* proofs: for instance it is hard to convince oneself that $\lambda\alpha.\alpha$ is a proof of $s(x) = s(y) \Rightarrow x = y$.

3 Pure Pattern Type Systems

P^2TS were introduced by Barthe, Cirstea, Kirchner and Liquori in [2]. In this section, we recall the syntax of P^2TS, their evaluation rules and their type systems.

$$\frac{}{\Gamma \vdash_{\cong} \alpha : \psi} \ (Ax) \text{ if } \alpha : \varphi \in \Gamma \text{ and } \varphi \cong \psi$$

$$\frac{\Gamma, \alpha : \varphi \vdash_{\cong} \pi : \psi}{\Gamma \vdash_{\cong} \lambda\alpha.\pi : \chi} \ (\Rightarrow I) \text{ if } \chi \cong \varphi \Rightarrow \psi$$

$$\frac{\Gamma \vdash_{\cong} \pi : \chi \qquad \Gamma \vdash_{\cong} \pi' : \varphi}{\Gamma \vdash_{\cong} \pi\pi' : \psi} \ (\Rightarrow E) \text{ if } \chi \cong \varphi \Rightarrow \psi$$

$$\frac{\Gamma \vdash_{\cong} \pi : \varphi}{\Gamma \vdash_{\cong} \lambda x.\pi : \psi} \ (\forall I) \text{ if } \psi \cong \forall x.\varphi \text{ and } x \notin \mathcal{FV}(\Gamma)$$

$$\frac{\Gamma \vdash_{\cong} \pi : \psi}{\Gamma \vdash_{\cong} \pi t : [t/x]\varphi} \ (\forall E) \text{ if } \psi \cong \forall x.\varphi$$

Figure 2. Proof-terms *à la* Dowek-Werner

3.1 P^2TS: dynamic semantics

Notations Syntactic equivalence of terms will be denoted by $\equiv$. If a substitution θ has domain $X_1 \dots X_n$ and $\forall i, \theta(X_i) \equiv A_i$, we will also write it $[A_1/X_1 \dots A_n/X_n]$.

We will assume that X, Y, Z are variables; A, B, C are terms; P, Q are patterns; a, f, g are constants; Φ, Ψ are types; Ξ is an atomic type. Moreover, we will use: θ for a substitution; Γ, Δ for contexts; Σ for a signature.

The calculus The syntax of P^2TS extends that of the typed λ-calculus with structures and patterns [2]. Several choices can be made for the set of patterns P; in Section 6, we will see that we need patterns to be either algebraic terms or a certain class of product types.

$Signatures$	$\Sigma, \Delta ::= \emptyset \mid \Sigma, f : A$
$Contexts$	$\Gamma ::= \emptyset \mid \Gamma, X : A$
$Patterns$	$P \subseteq A$
$Terms$	$A ::= f \mid X \mid \lambda(P : \Delta).A \mid \Pi(P : \Delta).A$
	$\mid [P \ll_{\Delta} A]A \mid A\,A \mid A; A$

A term $\lambda(P : \Delta).A$ is an *abstraction* with pattern P, body A and context Δ. A term $[P \ll_{\Delta} B]A$ is a *delayed matching constraint* with pattern P, body A, argument B and context Δ. A term $A\,B$ is an *application*. The usual algebraic notation of a term is currified, *e.g.* $f(A_1, \dots, A_n) \triangleq f\,A_1 \cdots A_n$. A term $(A; B)$ is called a *structure* with elements A and B. A term $\Pi(P : \Delta).A$ is a *dependent product*, and will be used as a type.

P^2TS feature pattern abstractions whose application requires solving matching problems, which we will denote $P \ll A$. A solution of $P \ll A$ is a substitution $\theta_{(P \ll A)}$ such that $\mathcal{D}om(\theta_{(P \ll A)}) = \mathcal{FV}(P)$ and $P\theta_{(P \ll A)} = A$. As we will see in Section 6, since some patterns are product

$$\begin{array}{lrcll}
(\rho) & (\lambda(P:\Delta).A)\,B & \to_\rho & [P \ll_\Delta B]A & \\
(\sigma) & [P \ll_\Delta B]A & \to_\sigma & A\theta_{(P\ll B)} & \text{if } \exists\,\theta_{(P\ll B)} \\
(\delta) & (A;B)\,C & \to_\delta & A\,C;B\,C &
\end{array}$$

Figure 3. Top-level rules of P^2TS

types, the notion of free variables of P has to be defined accordingly, and the considered equality is α-conversion.

Extending Church's notation, the context Δ in $\lambda(P : \Delta).B$ (resp. $[P \ll_\Delta B]A$ or $\Pi(P : \Delta).B$) contains the type declarations of the free variables appearing in the pattern P, *i.e.* $\mathcal{D}om(\Delta) = \mathcal{F}V(P)$.

The top-level rules are presented in Fig. 3. The most important is the (σ) rule, which consists in solving the matching equation $P \ll B$ and applying the obtained substitution (if it exists) to the the term A. If no solution exists, the (σ) rule is not fired and the term $[P \ll_\Delta B]A$ is not reduced. As usually, $\mapsto_{\rho\sigma\delta}$ denotes the congruent closure of $\to_\rho \cup \to_\sigma \cup \to_\delta$, and $\mapsto\!\!\!\to_{\rho\sigma\delta}$ (resp. $=_{\rho\sigma\delta}$) is defined as the reflexive and transitive (resp. reflexive, symmetric and transitive) closure of $\mapsto_{\rho\sigma\delta}$.

3.2 P^2TS: static semantics

P^2TS were designed in order to provide a calculus with pattern matching capabilities enjoying strongly normalization. This is achieved by a richer type system integrating patterns into types, reminiscent of a dependent types discipline. In Fig. 4, we give the main inference rules, taken from [14]. For a detailed explanation of these rules, the interested reader can refer to the alternative presentation of [2]. We only recall here the main modifications with respect to traditional Pure Type Systems.

- (Abs) is inspired by the abstraction typing rule for dependent types, with the difference that the pattern P appears in the Π-type;
- (Appl) types an application with a delayed matching constraint, which may be resolved later using (Conv);
- (Match) is derived from (Appl) and (Abs) so that subject reduction holds;
- (MSort) and (Prod) regulate the formation of product types, depending on the allowed couples (s_1, s_2). They ensure that the pattern and the body of the product are typable in the extended context Γ, Δ.

Remark 1 (Product formation) *Notice that s_1 is a common sort for all the free variables of P, which ensures that only arguments of sort s_1 will be passed to a function typed with a rule (s_1, s_2).*

$$\frac{\Sigma,\Gamma \vdash A : \Psi \qquad \Sigma,\Gamma \vdash \Phi : s \qquad \Phi =_{\rho\sigma\delta} \Psi}{\Sigma,\Gamma \vdash A : \Phi}\ \text{(Conv)}$$

$$\frac{\Sigma,\Gamma,\Delta \vdash A : \Phi \qquad \Sigma,\Gamma \vdash \Pi(P:\Delta).\Phi : s}{\Sigma,\Gamma \vdash \lambda(P:\Delta).A : \Pi(P:\Delta).\Phi}\ \text{(Abs)}$$

$$\frac{\Sigma,\Gamma \vdash A : \Pi(P:\Delta).\Phi \qquad \Sigma,\Gamma \vdash [P \ll_\Delta B]\Phi : s}{\Sigma,\Gamma \vdash A\,B : [P \ll_\Delta B]\Phi}\ \text{(Appl)}$$

$$\frac{\Sigma,\Gamma,\Delta \vdash A : \Phi \qquad \Sigma,\Gamma \vdash [P \ll_\Delta B]\Phi : s}{\Sigma,\Gamma \vdash [P \ll_\Delta B]A : [P \ll_\Delta B]\Phi}\ \text{(Match)}$$

$$\frac{\begin{array}{c}\forall (X{:}\Psi) \in \Delta,\ \ \Sigma,\Gamma,\Delta \vdash \Psi : s_1 \\ \Sigma,\Gamma,\Delta \vdash P : \Psi_0 \qquad \Sigma,\Gamma,\Delta \vdash \Phi : s_2\end{array}}{\Sigma,\Gamma \vdash \Pi(P:\Delta).\Phi : s_2}\ \text{(Prod)}$$

$$\frac{\begin{array}{c}\forall (X{:}\Psi) \in \Delta,\ \ \Sigma,\Gamma,\Delta \vdash \Psi : s_1 \\ \Sigma,\Gamma,\Delta \vdash P : \Psi_0 \\ \Sigma,\Gamma \vdash B : \Psi_0 \qquad \Sigma,\Gamma,\Delta \vdash \Phi : s_2\end{array}}{\Sigma,\Gamma \vdash [P \ll_\Delta B]\Phi : s_2}\ \text{(MSort)}$$

Figure 4. The typing rules of P^2TS

In [14], we proved strong normalization of the simply typed calculus, using a reduction-preserving encoding from P^2TS into System Fω:

Theorem 1 (Strong normalization of typable terms)
Every term typable using only the product rule $(*, *)$ is strongly normalizing.

Since we will also use the product rule $(*, \Box)$ to type our proof-terms, let us extend this result:

Theorem 2 (Strong normalization of ρP)
Every term typable using only the product rules $(*, *)$ and $(*, \Box)$ is strongly normalizing.

Proof: Following the lines of [11], we can define a type-erasure and reduction-preserving map from ρP-terms to $\rho_\to$-terms. The full proof is in Appendix A. $\Box$

We will use the product rule $(\Box, *)$ too, although in a quite limited way. Since this rule introduces impredicativity in the system, it is significantly harder to prove normalization; however, we conjecture that the encoding used for theorem 1 can be extended to any Pure Pattern Type System, the difficulty being essentially technical.

Conjecture 1 (Strong normalization of ρC)
Every term typable using the four product rules $(*, *)$, $(*, \Box)$, $(\Box, *)$ and $(\Box, \Box)$ is strongly normalizing.

4 P^2TS proof terms for deduction modulo

Since P^2TS include the λ-calculus and permit to express rewrite rules, we will use them to define proof-terms for deduction modulo, where the subterms inherited from the λ-calculus will account for logical inference rules, and the subterms featuring matching will track conversion steps. In this section we make precise which class of ρ-terms we will use.

4.1 An alternative type system

A problem arising when using dependent types to represent first-order logic is that a term A such that $\vdash A : *$ may represent either a proposition or a set. For better readability, we follow the approach proposed by Berardi [3] and implemented in the Coq proof assistant [13]: we use three sorts $*^s, *^p, \Box$, where $*^s$ and $*^p$ are intended to represent respectively sets and propositions. The sort hierarchy is now given by the axioms $\vdash *^s : \Box$ and $\vdash *^p : \Box$. The allowed product rules for a simple type system are

$$\{(*^s, *^s)\ ,\ (*^s, *^p)\ ,\ (*^p, *^p)\}$$

Finally, types depending on terms can be created using the product rule $(*^s, \Box)$ and terms depending on types require the rule $(\Box, *^p)$.

Moreover, we will use the following the notational conventions:

- φ, ψ, χ are such that $\vdash \varphi : *^p$, *i.e.* they represent propositions;
- μ, ν are such that $\vdash \mu : *^s$, *i.e.* they represent sets;
- variables α, β and terms π, π' are such that $\vdash \alpha : \varphi : *^p$, *i.e.* they represent proofs;
- variables x, y and terms t, s are such that $\vdash x : \mu : *^s$, *i.e.* they represent algebraic terms (and functions) of first-order logic.

It is easy to see that type inhabitation and normalization in this type system are equivalent to the ones in ρP, by the erasure function $*^s \mapsto *$ and $*^p \mapsto *$. Some variants of these PTS and morphisms between them have been studied by Geuvers [10]. Now we can explain how terms and propositions of first-order logic are represented in P^2TS.

Definition 2 (First-order logic represented in P^2TS) *We can consider a many-sorted signature, whose sorts are not to be mistaken for $*^s$, $*^p$ and $\Box$.*

- *For each sort s of the signature we take an atomic type μ such that $\vdash \mu : *^s$.*
- *For each variable x of sort s we take a variable x such that $\vdash x : \mu$.*
- *For each n-ary function $f : s_1, \ldots, s_n \mapsto s$ we take a constant symbol f such that $\vdash f : \Pi x_1{:}\mu_1 \ldots \Pi x_n{:}\mu_n.\mu : *^s$*
- *For each n-ary predicate symbol p over $s_1 \times \ldots \times s_n$ we take a constant symbol p such that $\vdash p : \Pi x_1{:}\mu_1 \ldots \Pi x_n{:}\mu_n.*^p : \Box$*

Terms and propositions are then represented as follows:

$$\begin{array}{rcll}
[\![x]\!] & \triangleq & x & \\
[\![f(t_1,\ldots,t_n)]\!] & \triangleq & f\,[\![t_1]\!]\ \ldots\ [\![t_n]\!] & \\
[\![p(t_1,\ldots,t_n)]\!] & \triangleq & p\,[\![t_1]\!]\ \ldots\ [\![t_n]\!] & \\
[\![\varphi \Rightarrow \psi]\!] & \triangleq & \Pi\alpha{:}[\![\varphi]\!].[\![\psi]\!] & \textit{where } \alpha \textit{ is fresh} \\
[\![\forall x.\varphi]\!] & \triangleq & \Pi x{:}\mu.[\![\varphi]\!] & \textit{where } \mu \textit{ is} \\
 & & & \textit{the sort of } x
\end{array}$$

It easy to see that for any term t of sort s we have $\vdash [\![t]\!] : \mu : *^s$ and for any proposition φ we have $\vdash [\![\varphi]\!] : *^p$.

4.2 Congruence on propositions

To account for the conversion steps, we need a dedicated operator Rew^p which will take a proposition φ, a proof-term of that proposition and apply a rewrite rule R to φ. Since the pattern appears in the type of a rewrite rule R, we need such an operator for each possible pattern l appearing in the rewrite rules:

$$\mathsf{Rew}^p_l : \Pi\varphi{:}\, *^p\ .\ \Pi R{:}(\Pi(l{:}*^p).*^p)\ .\ \Pi\alpha{:}\varphi\ .\ (R\varphi)$$

Given a proposition φ with proof π, the proposition φ' obtained by rewriting φ with a rule $l \to r$ has the proof-term $\mathsf{Rew}^p_l\ \varphi\ (\lambda l.r)\ \pi$. Indeed, the type of this term is $[l \ll \varphi]r$, which is convertible to φ'.

Each of the operators Rew^p_l is just a new constant to be added to the signature Σ. If the term rewriting system $\mathcal{R}$ defining the congruence $\cong$ is finite, then the signature remains finite too.

Typing the operators Rew^p requires the use of two product rules: $(*^s, \Box)$ (for the type of R) and $(\Box, *^p)$ (for abstracting on φ and R). Thus, we are working in the type system $\rho P2$, featuring dependent types and impredicativity of sort $*^p$. Cut elimination results will hold only under the assumption of strong normalization, which has not been proved for this system; however, it is not our only concern here. Alternatively, we could have defined an impredicative version of Rew^p, parameterized not only by a pattern but by a whole rewrite rule, losing some generality. However, as we will see in the following subsection, impredicativity is essential for dealing with a congruence on terms.

4.3 Congruence on terms

We have seen how to keep track of the application of a rewrite rule at the top-level of a formula; in Section 5, we will see that it is indeed sufficient for rewriting a proposition at any position of a formula, by an appropriate expansion of the logical inference.

However, when it comes to rewriting an algebraic term in a formula (*i.e.* some argument of a predicate symbol p occuring in the formula), we need a way to determine the exact position where the rewrite rule has to be applied. We achieve this by defining a kind of *generalized Leibniz equality*, where the proof of equality of two terms is replaced by a rewrite rule:

$$\begin{array}{ll}\mathsf{Rew}_l^s: & \Pi\varphi{:}(\Pi(y{:}\mu).*^p)\ .\ \Pi x{:}\mu\ .\ \Pi R{:}(\Pi(l{:}\mu).\mu)\ .\\ & \Pi\alpha{:}\varphi(x)\ .\ (\varphi(Rx))\end{array}$$

where $\vdash l : \mu$. Then, given an algebraic term t that rewrites to u at top-level using a rewrite rule $l \rightarrow r$, given a proposition $\varphi(t)$ with proof π, the proposition $\varphi(u)$ has the proof-term $\mathsf{Rew}_l^s\ \varphi\ t\ (\lambda l.r)\ \pi$. Indeed, the type of this term is $\varphi([l \ll t]r)$, which is convertible to $\varphi(u)$.

The ability to take φ as an argument (using the product rule $(\Box, *^p)$) is essential: it allows one to apply a rewrite rule to an algebraic term occuring at *any* position in the proposition.

Example 4 *The inferences from examples 1 and 2 have respective ρ-proof-terms*

$$\lambda\alpha.\Big(\mathsf{Rew}_{e*x}^s\ \big(\lambda y.(y{=}e)\big)\ (e*e')\ \big(\lambda(e{*}x).x\big)\ (\alpha e)\Big)$$

and

$$\lambda\alpha.\left(\begin{array}{c}\mathsf{Rew}_{p(s(z))}^s\ \big(\lambda w.(a{=}w)\big)\ p(s(b))\ \big(\lambda p(s(z)).z\big)\\ \left(\begin{array}{c}\mathsf{Rew}_{s(x)=y}^p\ \big(s(a){=}s(b)\big)\\ \big(\lambda(s(x){=}y).(x{=}p(y))\big)\ \alpha\end{array}\right)\end{array}\right)$$

As a summary to this section, the P^2TS-terms we will use are given by the grammar in Fig. 5. Notice that, while this classification into algebraic terms and proof-terms is implicitly based on the type system, not every term produced by the given grammar is well-typed. Now that we have seen which proof-terms we will use, we can discuss some technical aspects: the next two sections explain respectively how we manipulate rewrite rules and which pattern matching problems we solve.

5 Reversed rules

In this section we study extensively the application of rules right-to-left: we show that it is sufficient for application of rules below logical connectors, and that it enables to express a form of proof genericity.

Algebraic terms	t	$::=$	$f \mid x \mid t\,t'$
Propositions	φ	$::=$	$p \mid \varphi\,t \mid \Pi\alpha{:}\varphi.\varphi \mid \Pi x{:}\mu.\varphi$
Rewrite rules	R	$::=$	$\lambda t.t \mid \lambda\varphi.\varphi$
Proof-terms	π	$::=$	$\alpha \mid \lambda\alpha.\pi \mid \pi\,\pi' \mid \lambda x.\pi \mid \pi\,t \mid$ $\mathsf{Rew}_l^p\ \varphi\ R\,\pi \mid \mathsf{Rew}_l^s\ \varphi\ t\ R\,\pi$

Figure 5. Proof-terms in P^2TS style

It can easily be seen that two-way application of rewrite rules is unavoidable: in deduction modulo, if ψ is provable and the only rule we have is $\varphi \rightarrow \psi$, then φ is provable. Now, supposing π is a proof-term for ψ, the only sensible way to build a proof-term for φ is $\mathsf{Rew}_\psi^p\ \psi\ (\lambda\psi.\varphi)\ \pi$, where the rewrite rule $\psi \rightarrow \varphi$ appears.

Though two-way application of rewrite rules is surprising, for our purposes it remains perfectly sound: remember we introduced a rewrite system in order to define a congruence by symmetric and transitive closure. Moreover, there can not be any problem related to termination (of the rewriting system) since a rewrite rule appears once in our proof-terms for *each* of its applications. The decidability of the congruence $\cong$ is not necessary anymore since we keep a witness for each conversion, so we can even consider rewriting systems that are not confluent nor terminating.

5.1 Application of a rewrite rule inside a formula

In a proof-term $\mathsf{Rew}_l^p\ \varphi\ R\ \pi$, the rewrite rule R is always applied to the head position of φ. Let us see how we can deal with non-atomic formulas. Suppose we have the following inference:

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash_\cong \varphi_1 \Rightarrow \varphi_2}}{\Gamma \vdash_\cong \psi_1 \Rightarrow \psi_2}\ (\cong)$$

where the conversion corresponds to application of a rewrite rule on one of the two subformulas. We can expand this derivation into another one proving the same statement from the same assumptions but using the conversion rule on a subformula only.

- if $\varphi_1 \mapsto_{\mathcal{R}} \psi_1$ and $\varphi_2 \equiv \psi_2$, we take

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma,\psi_1 \vdash_\cong \psi_1}\ (Ax)}{\Gamma,\psi_1 \vdash_\cong \varphi_1}\ (\cong) \qquad \dfrac{\vdots}{\Gamma,\psi_1 \vdash_\cong \varphi_1 \Rightarrow \varphi_2}}{\Gamma,\psi_1 \vdash_\cong \varphi_2}\ (\Rightarrow E)}{\Gamma \vdash_\cong \psi_1 \Rightarrow \varphi_2}\ (\Rightarrow I)$$

- if $\varphi_2 \mapsto_{\mathcal{R}} \psi_2$ and $\varphi_1 \equiv \psi_1$, we take

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, \varphi_1 \vdash_{\cong} \varphi_1}\ (Ax) \qquad \dfrac{\vdots}{\Gamma, \varphi_1 \vdash_{\cong} \varphi_1 \Rightarrow \varphi_2}}{\Gamma, \varphi_1 \vdash_{\cong} \varphi_2}\ (\Rightarrow E)}{\Gamma, \varphi_1 \vdash_{\cong} \psi_2}\ (\cong)}{\Gamma \vdash_{\cong} \varphi_1 \Rightarrow \psi_2}\ (\Rightarrow I)$$

In the first case, the new conversion rule has ψ_1 as an assumption and φ_1 as a conclusion: the rewrite rule $\varphi_1 \to \psi_1$ has to be applied right-to-left. Similar expansions of conversions can be used to decompose a formula with any other head connector.

5.2 Non-regular rules and proof genericity

An interesting feature of the framework appears with non-regular rewriting systems (*i.e.* where a rule $l \to r$ is such that $Var(r) \subsetneq Var(l)$). Indeed, when applying rules the reverse way, we create new unbound variables which can be freely instantiated later.

For instance, consider a predicate $last$ intended to express that a given element is the last of a given list. A way to define it is to take the congruence defined by the (non-regular) rewrite rule

$$last(y{::}(z{::}zz), x) \to last(z{::}zz, x)$$

and have a single axiom $LastAx : \forall x. last(x{::}nil, x)$ in the assumptions.

For better readability, let us write l for $last(y{::}(z{::}zz), x)$ and r for $last(z{::}zz, x)$. The encoded rewrite rule $\lambda l.r$ is closed, whereas in the reversed rule $\lambda r.l$ the variable y appears free. Let us see which proof-terms we can build in this framework.

- $LastAx\, b$ (corresponding to a $(\forall E)$) is a closed proof-term for $last(b{::}nil, b)$.

- $\mathsf{Rew}^p_r\ last(b{::}nil, b)\ (\lambda r.l)\ (LastAx\ b)$, obtained by one rewrite step, is a proof-term for $last(y{::}(b{::}nil), b)$ where y is a free variable, which can be seen as a *generic* proof over all possible y.

- $\lambda y.\Big(\mathsf{Rew}^p_r\ last(b{::}nil, b)\ (\lambda r.l)\ (LastAx\ b)\Big)$ is a closed (remember y appears in l) proof-term for $\forall y. last(y{::}(b{::}nil), b)$, obtained by a simple abstraction over y (corresponding to a $(\forall I)$).

- $\Big(\lambda y.\Big(\mathsf{Rew}^p_r\ last(b{::}nil, b)\ (\lambda r.l)\ (LastAx\ b)\Big)\Big)\ a$ is a closed proof-term for $last(a{::}(b{::}nil), b)$. If the outermost redex is reduced, the rewrite rule $\lambda r.l$ is replaced by $\lambda r.(l[a/y])$, which is a closed instance of it.

Similarly, one can build proof terms for an arbitrary number of elements in the list, either generic as in $\forall x_1 \ldots \forall x_n. last(x_1{::}(\ldots(x_n{::}(d{::}nil))), d)$, or ground as in $last(a_1{::}(\ldots(a_n{::}(d{::}nil))), d)$.

6 Pattern matching modulo α-conversion

In this section we discuss which class of patterns appear in our proof-terms, and how we should solve the corresponding matching problems.

6.1 Matching problems

A first immediate remark is that, according to Fig. 5, the deductive part of our proof-terms uses only variables as patterns. Non-trivial matching problems arise only with the rewrite rules given as arguments to Rew^p and Rew^s.

Matching on algebraic terms is hardly more complex: patterns can be any term, *i.e.* either a variable x or a composed term $f\, t_1 \ldots t_n$. In this case we only have to deal with standard syntactic matching:

- in an algebraic pattern, every variable is free;

- a substitution θ is a solution of $P \ll A$ if $P\theta \equiv A$.

For propositions, in deduction modulo, usually one considers only rules whose left-hand size is an atomic proposition $p(t_1, \ldots, t_n)$. In that case, matching remains syntactic. However, we have seen that rules have to be used from right to left too, so we may have to perform matching with a pattern which represents a (non-atomic) proposition.

We have encoded implication as a product type $\Pi\alpha{:}\varphi.\psi$ (where α does not occur in ψ) and universal quantification as $\Pi x{:}\mu.\varphi$ (where x may appear in φ). One immediately sees that for implication, the variable α is irrelevant; for quantification, the variable x can be renamed if each of its (free) occurrences in φ is simultaneously renamed the same way. Therefore, the only free variables appearing in a proposition pattern are inherited from atomic predicates and can be only algebraic variables.

The suitable matching theory uses α-conversion:

- the free variables of a pattern are defined inductively:

$$\begin{array}{rcl} \mathcal{FV}(p\, t_1 \ldots t_n)) & \triangleq & \bigcup \mathcal{V}ar(t_i) \\ \mathcal{FV}(\Pi\alpha{:}\varphi.\psi) & \triangleq & \mathcal{FV}(\varphi) \cup \mathcal{FV}(\psi) \\ \mathcal{FV}(\Pi x{:}\mu.\varphi) & \triangleq & \mathcal{FV}(\varphi) \setminus \{x\} \end{array}$$

- a substitution θ is a solution of $P \ll A$ if $P\theta =_\alpha A$

Example 5 (Non-atomic proposition patterns)

- $\Pi\beta{:}(x{\neq}0).(y{=}0)$ *represents* $x{\neq}0 \Rightarrow y{=}0$. *It matches* $\Pi\alpha{:}(t \neq 0).(t' = 0)$ *for any variable* α *and any terms* t *and* t'.

$$\begin{array}{rcll} f(t_1,\ldots,t_n) \ll g(s_1,\ldots,s_n) & \longrightarrow & t_1 \ll s_1 \wedge \ldots \wedge t_n \ll s_n & \text{if } f = g \\ p(t_1,\ldots,t_n) \ll q(s_1,\ldots,s_n) & \longrightarrow & t_1 \ll s_1 \wedge \ldots \wedge t_n \ll s_n & \text{if } p = q \\ \Pi(\alpha : P_1).P_2 \ll \Pi(\beta : Q_1).Q_2 & \longrightarrow & P_1 \ll Q_1 \;\; \wedge \;\; P_2 \ll Q_2 & \\ \Pi(x : \tau).P_{(x)} \ll \Pi(y : \sigma).Q_{(y)} & \longrightarrow & [z/y]P \ll [z/x]Q & \text{for a fresh variable } z \\ x \ll t \;\wedge\; x \ll t' & \longrightarrow & \bot & \text{if } t \neq t' \end{array}$$

Figure 6. A matching algorithm for propositions

- $\Pi y{:}int.(x{*}y = 0)$ *represents* $\forall y.(x{*}y = 0)$. *It matches* $\Pi z{:}int.(t{*}z = 0)$ *for any variable* z *and any term* t.

6.2 A matching algorithm

Matching modulo α-conversion can be achieved by syntactic decomposition and systematic renaming of bound variables. However, as pointed out in the previous subsection, the propositional variables are irrelevant since they cannot appear in the body of a Π-abstraction. In Fig. 6 we give a set of transformation rules such that:

- every matching problem $P \ll M$ has a unique normal form with respect to these rules;
- if it is empty, then P and M are identical up to α-conversion;
- if it is of the form $\bigwedge_{i \in I} x_i \ll t_i$ with $I \neq \emptyset$, then $\theta = [t_i/x_i]$ is the unique substitution such that $P\theta =_\alpha M$;
- otherwise, there is no match from P to M.

7 The isomorphism

Let us now discuss to which extent we have defined a Curry-Howard isomorphism for deduction modulo. In this section, we explicit the propositions-as-types and proofs-as-terms embedding, and we study which notion of cut elimination is modeled by reduction in our proof-term language.

We define two maps for transforming first-order logic propositions and terms into P^2TS algebraic terms (with sort $*^s$) and types (with sort $\Box$), and *vice versa*.

Definition 3 (Propositions-as-types)

1. $[\![\cdot]\!] : FOL \to P^2TS$ *is detailed in Def. 2. It can be extended to contexts by assigning a fresh name to each hypothesis:*

$$[\![\Gamma, \varphi]\!] \triangleq [\![\Gamma]\!], x_\varphi : [\![\varphi]\!]$$

2. $|\cdot| : P^2TS \to FOL$ *is an erasure map, defined the converse way:*

$$\begin{array}{rcl} |x| & \triangleq & x \\ |f\, t_1 \,\ldots\, t_n| & \triangleq & f(|t_1|,\ldots,|t_n|) \\ |p\, t_1 \,\ldots\, t_n| & \triangleq & p(|t_1|,\ldots,|t_n|) \\ |\Pi\alpha{:}\varphi.\psi| & \triangleq & |\varphi| \Rightarrow |\psi| \\ |\Pi x{:}\mu.\varphi| & \triangleq & \forall x.|\varphi| \\ |\Gamma, \alpha{:}\varphi| & \triangleq & |\Gamma|, |\varphi| \\ |\Gamma, x{:}\mu| & \triangleq & |\Gamma| \end{array}$$

Theorem 3 (Proofs-as-terms)

1. *Every type inhabited by a proof-term from Fig. 5 represents a provable proposition:*

$$\Gamma \vdash \pi : \varphi \;\Rightarrow\; |\Gamma| \vdash_{\cong} |\varphi|$$

2. *Every provable proposition is represented as a type inhabited by a proof-term:*

$$\Gamma \vdash_{\cong} \varphi \;\Rightarrow\; \exists \pi,\;\; [\![\Gamma]\!] \vdash \pi : [\![\varphi]\!]$$

Proof:

1. The translation from typed proof-terms to proofs is quite straightforward: variables α are mapped to axioms, λ-abstractions on implication introductions, *etc*.
2. To build a proof-term for a provable proposition, the derivation has to be transformed, because we only apply conversion rule by rule and at the top-level position of the proposition. Thus, a first step is to have exactly one conversion rule for each application of a rewrite rule. The second step is to "expand" the conversion steps so that conversion occurs at top-level. Finally, we can build a proof-term with a simple translation mapping axioms to variables, *etc*.

The full proof can be found in Appendix A. $\Box$

As pointed out in [9], the main drawback of the explicit conversion rule $(\cong)$ is that a cut is not "local" in a proof: it is formed by an introduction, an arbitrary number of conversions, and an elimination. For instance, with the congruence $p \cong q$ the proof

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p \vdash_{\cong} p}\;(Ax)}{\vdash_{\cong} p \Rightarrow p}\;(\Rightarrow I)}{\vdash_{\cong} q \Rightarrow p}\;(\cong) \qquad \dfrac{\vdots}{\vdash_{\cong} q}}{\vdash_{\cong} p}\;(\Rightarrow E)$$

can be reduced to

$$\dfrac{\dfrac{\vdots}{\vdash_{\cong} q}}{\vdash_{\cong} p}\;(\cong)$$

In our proof-terms, we chose to explicitly account for the conversions. In some cases, the cut will still appear as a redex: for instance, the first derivation above has proof-term $\Big(\lambda\alpha{:}q.\big((\lambda\beta{:}p.\beta)(\mathsf{Rew}^p_q\ q\ (\lambda q.p)\ \alpha)\big)\Big)\pi$ where π is a proof-term for q. This term reduces to $\mathsf{Rew}^p_q\ q\ (\lambda q.p)\ \pi$, which is indeed a proof-term for the reduced derivation.

Unfortunately, a cut with too many nested conversions may not appear as a redex in the proof-term. The situation occurs typically when a conversion step is followed by its opposite: a proper abstraction $\lambda\alpha.\pi$ (corresponding to an implication introduction) can be hidden below two (or more) constructs Rew^p, disabling any possible redex involving this function. These "unreducible cuts" can be considered as a symptom of the fact that our proof-term language lacks some reduction rules.

Conjecture 2 (Cuts-as-redexes) *With additional reduction rules over the conversion constructions such as*

$$\mathsf{Rew}^p_l \varphi\ (\lambda l.r)\ \big(\mathsf{Rew}^p_{l'}\ \varphi'\ (\lambda l'.r')\ \pi\big) \ \mapsto\ \pi \quad \textit{if}\ \ l{=}r' \wedge l'{=}r$$

every cut in a proof is modeled as a redex in its proof-term.

In the next section, we will see that, even if a reducible proof is assigned a normal proof-term, we still have a simple way of detecting this cut, so the notion of normal proof is not lost in our isomorphism.

8 Connections with other formalisms

A great interest of the proposed proof-term language resides in the various relations one can establish with other formalisms and tools. In this section we show how to extract the deductive content and the computational content from proof-terms; we also discuss the modalities of an actual implementation.

8.1 Proof structure

The erasure map $|\cdot|$ transforms a P^2TS-proof-term into the λ-proof-term which represents the same proof with implicit conversion, as detailed in Fig. 2. Therefore, every P^2TS-term π such that $|\pi|$ is in normal form represents a proof in normal form, and cut elimination holds for the class of theories having a premodel, as shown by Dowek and Werner [9].

$$\begin{array}{rcl}
|\cdot| & : & P^2TS \longrightarrow \Lambda \\
|x| & \triangleq & x \\
|\lambda x{:}\mu.\pi| & \triangleq & \lambda x{:}\mu.|\pi| \\
|\pi t| & \triangleq & |\pi||t| \\
|\lambda\alpha{:}\varphi.\pi| & \triangleq & \lambda\alpha{:}\varphi.|\pi| \\
|\pi\,\pi'| & \triangleq & |\pi|\,|\pi'| \\
|\mathsf{Rew}^p_l\ \varphi\ (\lambda l.r)\ \pi| & \triangleq & \pi \\
|\mathsf{Rew}^s_l\ \varphi\ t\ (\lambda l.r)\ \pi| & \triangleq & \pi
\end{array}$$

8.2 Congruence bookkeeping

The function $\mathcal{R}(\cdot)$ produces the set of rewrite rules used in a proof. If the congruence is known *a priori*, it can be used to check that the computations did not involve any unauthorized rewrite rule. It can also be used for characterizing a subset of the rewriting system that is required for one proof or one set of proofs, and drop unnecessary rules.

$$\begin{array}{rcl}
\mathcal{R}(\cdot) & : & \rho \longrightarrow \text{TRS} \\
\mathcal{R}(x) & \triangleq & \emptyset \\
\mathcal{R}(\lambda x.\pi) & \triangleq & \mathcal{R}(\pi) \\
\mathcal{R}(\pi t) & \triangleq & \mathcal{R}(\pi) \\
\mathcal{R}(\lambda\alpha.\pi) & \triangleq & \mathcal{R}(\pi) \\
\mathcal{R}(\pi\,\pi') & \triangleq & \mathcal{R}(\pi) \cup \mathcal{R}(\pi') \\
\mathcal{R}(\mathsf{Rew}^p_l\ \varphi\ (\lambda l.r)\ \pi) & \triangleq & \mathcal{R}(\pi) \cup \{l \rightarrow r\} \\
\mathcal{R}(\mathsf{Rew}^s_l\ \varphi\ t\ (\lambda l.r)\ \pi) & \triangleq & \mathcal{R}(\pi) \cup \{l \rightarrow r\}
\end{array}$$

8.3 Building P^2TS-terms

A first approach would be to modify only slightly the Coq environment in order to produce P^2TS-terms. The automation would remain very limited, since using a rewrite rule would require approximately the same work as using Leibniz equality.

A more promising idea is to use an external tool for computing and producing the corresponding proof-terms. A closely related instance of this proof paradigm is Nguyen's tool [12], which uses the rewrite-based language ELAN to check the equality of two terms relatively to a rewrite system and produce a proof-term for Coq. The user performs the interactive proof the usual way, with the possibility to call the tactic ElanRewrite at any point to normalize all first-order subterms of any goal. We could probably use the Coq/ELAN interface as a basis for an interactive prover producing P^2TS-proof-terms.

9 Conclusion and perspectives

We have seen how to use P^2TS terms to represent proofs in natural deduction modulo with an explicit account for conversion. Two slightly different constructs are used for conversion on terms and on propositions, and every rewrite rule used appears in the proof-term. The reduction relation of P^2TS captures only a fragment of cut elimination, so our representation could be used effectively for proof search only at the price of additional reduction rules. Still, pure λ-calculus proof-terms where every cut appears are embedded in P^2TS-proof terms and can be recovered by a simple erasure map.

The usual proof construction techniques of environments like Coq (interactive proof with or without calls to an external tool such as ELAN or C*i*ME) are easily adapted for producing P^2TS proof-terms. Conversely, the information about conversion steps included in the proof-terms makes proof-checking easier: there is no need anymore for a procedure deciding if two propositions are convertible, and the considered congruence can even be undecidable. The main interest of our representation is also its principal limit: proof-terms are quite large, which could be a problem when dealing with high scale proofs.

An immediate application of our work would be to extend the Coq environment in order to produce and manipulate P^2TS proof-terms. In Barendregt and Barendsen's terminology, this new proof assistant would be classified as autarkic with a considerable latitude in using the skeptical style, *i.e.* able to rely on an external system for computation if it provides a suitable trace.

As stated in the discussion about cut elimination, a more fundamental perspective would be to see which additional reduction rules are needed to represent every cut. The strong normalization of the extended reduction relation should be studied then. On this line of research, we could possibly find a new characterization of rewrite systems for which cut elimination holds, extending the works of Dowek and Werner on that subject.

Acknowledgements The author would like to thank his advisors for asking him so many times to push researches on the present topic before the main ideas came forth.

References

[1] H. Barendregt and E. Barendsen. Autarkic computations in formal proofs. *Journal of Automated Reasoning*, 28(3):321–336, Apr. 2002.

[2] G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure Patterns Type Systems. In *Principles of Programming Languages - POPL2003, New Orleans, USA*. ACM, Jan. 2003.

[3] S. Berardi. Towards a mathematical analysis of the coquand-huet calculus of constructions and the other systems in barendregt's cube. Technical report, Dept. Computer Science, Carnegie Mellon University and Dipartimento Matematica, Universita di Torino, Italy, 1988.

[4] F. Blanqui. Definitions by rewriting in the calculus of constructions. In *Logic in Computer Science*, pages 9–18, 2001.

[5] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95 – 120, 1988.

[6] T. Coquand and C. Paulin. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *COLOG-88. Proceedings of International Conference on Computer Logic, Tallinn, Estonia*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer-Verlag, 1990.

[7] N. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In S. Verlag, editor, *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 29 – 61, Versailles, 1970.

[8] G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 31(1):33–72, Nov 2003.

[9] G. Dowek and B. Werner. Proof normalization modulo. *Journal of Symbolic Logic*, 68(4):1289–1316, 2003.

[10] H. Geuvers. *Logics and type systems*. PhD thesis, Nijmegen University, 1993.

[11] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Second Symposium of Logic in Computer Science*, pages 194 – 204, Ithaca, N. Y., 1987. IEEE, Washington DC.

[12] Q.-H. Nguyen, C. Kirchner, and H. Kirchner. External rewriting for skeptical proof assistants. *Journal of Automated Reasoning*, 29(3-4):309–336, 2002.

[13] LogiCal project. *The* Coq *proof assistant*. INRIA, Rocquencourt, France, version 8.0 edition, 2004.

[14] B. Wack. The simply-typed pure pattern type system ensures strong normalization. In J.-J. Lévy, E. Mayr, and J. Mitchell, editors, *3rd International Conference on Theoretical Computer Science*, pages 633 – 646, Toulouse, France, August 2004. IFIP, Kluwer Academic Publishers.

A Proofs (for referees only)

Theorem 2 (Strong normalization of ρP)
Every term typable using only the product rules $(*, *)$ and $(*, \Box)$ is strongly normalizing.

Proof: Following the lines of [11], we define a translation τ of sorts and types and a type-erasure and reduction-preserving map $|\cdot|$ from ρP-terms to $\rho_{\rightarrow}$-terms. We use a particular constant 0 with $\vdash_{\rho_{\rightarrow}} 0 : *$ and a family of constants $\{\pi_P \mid P \text{ a pattern}\}$. A constant π_P has type $0 \rightarrow \ldots \rightarrow 0 \rightarrow (\Pi P{:}\tau(\Delta).0) \rightarrow 0$ with n times 0 if P has n free variables with types given by Δ (arrows stand for the usual abbreviation of Π-abstractions).

$$\begin{aligned}
\tau(\Box) &\triangleq 0\\
\tau(*) &\triangleq 0\\
\tau(x) &\triangleq x \qquad \text{if } \vdash_{\rho P} x : A : \Box\\
\tau(f) &\triangleq f \qquad \text{if } \vdash_{\rho P} f : A : \Box\\
\tau(\Pi P{:}\Delta.B) &\triangleq \Pi P{:}\tau(\Delta).\tau(B)\\
\tau(\lambda P{:}\Delta.B) &\triangleq \tau(B)\\
\tau([P \ll_{\Delta} A]B) &\triangleq [P \ll_{\tau(\Delta)} |A|]\tau(B)\\
\tau(A\,B) &\triangleq \tau(A)
\end{aligned}$$

$$\begin{aligned}
|x| &\triangleq x\\
|f| &\triangleq f\\
|\Pi P{:}\Delta.B| &\triangleq \pi_{\tau(\Delta)}\, |A_1| \ldots |A_n|\, (\lambda P{:}\tau(\Delta).|B|)\\
&\qquad \text{if } \Delta \equiv x_1{:}A_1 \ldots x_n{:}A_n\\
|\lambda P{:}\Delta.B| &\triangleq \lambda P{:}\tau(\Delta).\\
&\quad \big((\lambda y_1{:}0 \ldots \lambda y_n{:}0.|B|)\, |A_1| \ldots |A_n|\big)\\
&\qquad \text{if } \Delta \equiv x_1{:}A_1 \ldots x_n{:}A_n\\
|A\,B| &\triangleq |A|\,|B|\\
|[P \ll_{\Delta} B]C| &\triangleq [P \ll_{\tau(\Delta)} |B|]\\
&\quad \big((\lambda y_1{:}0 \ldots \lambda y_n{:}0.|C|)\, |A_1| \ldots |A_n|\big)
\end{aligned}$$

The correctness of these functions is given by the two following lemmas, which are easily proved by systematic inspection of all cases:

1. If $\Gamma \vdash_{\rho P} A : B$, then $\tau(\Gamma) \vdash_{\rho_{\rightarrow}} |A| : \tau(B)$
2. If $A \longmapsto_{\rho\sigma\delta} A'$, then $|A| \twoheadrightarrow_{\rho\sigma\delta}{}^{+} |A'|$

$\Box$

Theorem 3 (Proofs-as-terms)

1. *Every type inhabited by a proof-term from Fig. 5 represents a provable proposition:*

$$\Gamma \vdash \pi : \varphi \;\Rightarrow\; |\Gamma| \vdash_{\cong} |\varphi|$$

2. *Every provable proposition is represented as a type inhabited by a proof-term:*

$$\Gamma \vdash_{\cong} \varphi \;\Rightarrow\; \exists \pi,\;\; [\![\Gamma]\!] \vdash \pi : [\![\varphi]\!]$$

Proof:

1. This direction is quite straightforward: we proceed by induction over the structure of π.

   $\pi \equiv \alpha$: then $(\alpha{:}\varphi) \in \Gamma$, so $|\varphi| \in |\Gamma|$ and immediately we have

$$\frac{}{|\Gamma| \vdash_{\cong} |\varphi|}\;(Ax)$$

   $\pi \equiv \lambda\alpha.\pi'$: then $\varphi \equiv \Pi\alpha{:}\varphi_1.\varphi_2$, the type derivation ends with (Abs) and has $\Gamma, \alpha{:}\varphi_1 \vdash \pi' : \varphi_2$ as an assumption. By induction we have $|\Gamma|, |\varphi_1| \vdash_{\cong} |\varphi_2|$ so we have

$$\frac{|\Gamma|, |\varphi_1| \vdash_{\cong} |\varphi_2|}{|\Gamma| \vdash_{\cong} |\varphi_1| \Rightarrow |\varphi_2|}\;(\Rightarrow I)$$

   and indeed $|\Pi\alpha{:}\varphi_1.\varphi_2| = |\varphi_1| \Rightarrow |\varphi_2|$.

   $\pi \equiv \lambda x.\pi'$: then $\varphi \equiv \Pi x{:}\mu.\varphi_1$, the type derivation ends with (Abs) and has $\Gamma, x{:}\mu \vdash \pi' : \varphi_1$ as an assumption. By induction we have $|\Gamma| \vdash_{\cong} |\varphi_1|$ where $x \notin \mathcal{FV}(\Gamma)$ so we have

$$\frac{|\Gamma| \vdash_{\cong} |\varphi_1|}{|\Gamma| \vdash_{\cong} \forall x.|\varphi_1|}\;(\forall I)$$

   and indeed $|\Pi x{:}\mu.\varphi_1| = \forall x.|\varphi_1|$

   $\pi \equiv \pi'\,\pi''$: then the type derivation ends with (App) and has $\Gamma \vdash \pi' : \Pi\alpha{:}\psi.\varphi$ and $\Gamma \vdash \pi'' : \psi$ as assumptions. By induction we have $|\Gamma| \vdash_{\cong} |\Pi\alpha{:}\psi.\varphi|$ and $|\Gamma| \vdash_{\cong} |\psi|$ so we have

$$\frac{|\Gamma| \vdash_{\cong} |\psi| \Rightarrow |\varphi| \quad |\Gamma| \vdash_{\cong} |\psi|}{|\Gamma| \vdash_{\cong} |\varphi|}\;(\Rightarrow E)$$

   since $|\Pi\alpha{:}\psi.\varphi| = |\psi| \Rightarrow |\varphi|$.

   $\pi \equiv \pi'\,t$: then the type derivation ends with (App) and has $\Gamma \vdash \pi' : \Pi x{:}\mu.\psi$ and $\Gamma \vdash t : \mu$ as assumptions, with $\varphi \equiv \psi[t/x]$. By induction we have $|\Gamma| \vdash_{\cong} |\Pi x{:}\mu.\psi|$ so we have

$$\frac{|\Gamma| \vdash_{\cong} \forall x.|\psi|}{|\Gamma| \vdash_{\cong} |\varphi|}\;(\forall E)$$

   since $|\Pi x{:}\mu.\varphi| = \forall x.|\varphi|$

$\pi \equiv \mathsf{Rew}^p_l \varphi'(\lambda l.r)\pi'$: then $\varphi \equiv [l \ll \varphi']r$ and the type derivation ends with various (App) and (Conv) rules, among which the assumption $\Gamma \vdash \pi' : \varphi'$ can be tracked. If the matching problem $l \ll \varphi'$ has no solution, then φ does not represent a proposition and π is not an interesting proof-term. Otherwise, $\varphi =_{\beta\sigma} r\theta_{(l\ll\varphi')}$ and by induction $|\Gamma| \vdash_{\cong} |\varphi'|$ so we have:

$$\frac{|\Gamma| \vdash_{\cong} |\varphi'|}{|\Gamma| \vdash_{\cong} |\varphi|}\ (\cong) \quad \text{using the rewrite rule } l \to r$$

$\pi \equiv \mathsf{Rew}^s_l \varphi' t(\lambda l.r)\pi'$: then $\varphi \equiv \varphi'([l \ll t]r)$ and the type derivation ends with various (App) and (Conv) rules, among which the assumption $\Gamma \vdash \pi' : \varphi'(t)$ can be tracked. If the matching problem $l \ll t$ has no solution, then φ does not represent a proposition and π is not an interesting proof-term. Otherwise, $\varphi =_{\beta\sigma} \varphi'(r\theta_{(l\ll t)})$ and by induction $|\Gamma| \vdash_{\cong} |\varphi'(t)|$ so we have:

$$\frac{|\Gamma| \vdash_{\cong} |\varphi'(t)|}{|\Gamma| \vdash_{\cong} |\varphi|}\ (\cong) \quad \text{using the rewrite rule } l \to r$$

2. We proceed in three steps.

The first step simply consists into making the congruence steps appear clearly: in the derivation, rule $(\cong)$ occurs for one (and only one) application (forward or backward) of a rewrite rule.

Then "expand" the congruence steps as shown in Section 5: supposing we have $\dfrac{\Gamma \vdash_{\cong} \psi}{\Gamma \vdash_{\cong} \varphi}\ (\cong)$, we distinguish cases over the conversion step between ψ and φ.

- if $\varphi \to_{\mathcal{R}} \psi$ or $\psi \to_{\mathcal{R}} \varphi$ at the top-level position, we proceed to the next step.
- if $\varphi \equiv \varphi_1 \Rightarrow \varphi_2$ with $\varphi_1 \cong \psi_1$ and $\psi \equiv \psi_1 \Rightarrow \varphi_2$, then we expand the conversion step into

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, \psi_1 \vdash_{\cong} \psi_1}\ (Ax)}{\Gamma, \psi_1 \vdash_{\cong} \varphi_1}\ (\cong) \qquad \Gamma, \psi_1 \vdash_{\cong} \varphi_1 \Rightarrow \varphi_2}{\Gamma, \psi_1 \vdash_{\cong} \varphi_2}\ (\Rightarrow E)}{\Gamma \vdash_{\cong} \psi_1 \Rightarrow \varphi_2}\ (\Rightarrow I)$$

  and recursively treat the conversion $\psi_1 \cong \varphi_1$.
- if $\varphi \equiv \varphi_1 \Rightarrow \varphi_2$ with $\varphi_2 \cong \psi_2$ and $\psi \equiv \varphi_1 \Rightarrow \psi_2$, then we expand the conversion step into

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, \varphi_1 \vdash_{\cong} \varphi_1}\ (Ax) \qquad \Gamma, \varphi_1 \vdash_{\cong} \varphi_1 \Rightarrow \varphi_2}{\Gamma, \varphi_1 \vdash_{\cong} \varphi_2}\ (\Rightarrow E)}{\Gamma, \varphi_1 \vdash_{\cong} \psi_2}\ (\cong)}{\Gamma \vdash_{\cong} \varphi_1 \Rightarrow \psi_2}\ (\Rightarrow I)$$

  and recursively treat the conversion $\varphi_2 \cong \psi_2$.
- if $\varphi \equiv \forall x.\varphi_1$ with $\varphi_1 \cong \psi_1$ and $\psi \equiv \forall x.\psi_1$, then we expand the conversion step into

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash_{\cong} \forall x.\psi_1}{\Gamma \vdash_{\cong} \psi_1}\ (\forall E)}{\Gamma \vdash_{\cong} \varphi_1}\ (\cong)}{\Gamma \vdash_{\cong} \forall x.\varphi_1}\ (\forall I)$$

  and recursively treat the new conversion $\psi_1 \cong \varphi_1$.

In the third step, we translate inference rules into proof-terms. The translation is roughly the reverse to the one detailed above for provability of inhabited propositions:

- axioms are translated into variables;
- introduction of $\Rightarrow$ is translated into λ-abstraction over a variable α;
- introduction of $\forall$ is translated into λ-abstraction over a variable x;
- elimination of $\Rightarrow$ and $\forall$ are translated into application;
- conversion over proposition (resp. term) is translated into a Rew^p_l (resp. Rew^s_l) construction. For terms, it is always possible; for propositions it is possible since conversions occur at top-level only.

$\square$