Rewriting Modulo β in the $\lambda\Pi$-Calculus Modulo

Ronan Saillard
MINES ParisTech, PSL Research University, France
ronan.saillard@mines-paristech.fr

The $\lambda\Pi$-calculus Modulo is a variant of the λ-calculus with dependent types where β-conversion is extended with user-defined rewrite rules. It is an expressive logical framework and has been used to encode logics and type systems in a shallow way. Basic properties such as subject reduction or uniqueness of types do not hold in general in the $\lambda\Pi$-calculus Modulo. However, they hold if the rewrite system generated by the rewrite rules together with β-reduction is confluent. But this is too restrictive. To handle the case where non confluence comes from the interference between the β-reduction and rewrite rules with λ-abstraction on their left-hand side, we introduce a notion of rewriting modulo β for the $\lambda\Pi$-calculus Modulo. We prove that confluence of rewriting modulo β is enough to ensure subject reduction and uniqueness of types. We achieve our goal by encoding the $\lambda\Pi$-calculus Modulo into Higher-Order Rewrite System (HRS). As a consequence, we also make the confluence results for HRSs available for the $\lambda\Pi$-calculus Modulo.

1 Introduction

The $\lambda\Pi$-calculus Modulo is a variant of the λ-calculus with dependent types ($\lambda\Pi$-calculus or LF) where β-conversion is extended with user-defined rewrite rules. Since its introduction by Cousineau and Dowek [8], it has been used as a logical framework to express different logics and type systems. A key advantage of rewrite rules is that they allow designing *shallow* embeddings, that is embeddings that preserve the computational content of the encoded system. It has been used, for instance, to encode functional Pure Type Systems [8], First-Order Logic [9], Higher-Order Logic [2], the Calculus of Inductive Constructions [4], resolution and superposition proofs [6], and the ς-calculus [7].

The expressive power of the $\lambda\Pi$-calculus Modulo comes at a cost: basic properties such as subject reduction or uniqueness of types do not hold in general. Therefore, one has to prove these properties for each particular set of rewrite rules considered. The usual way to do so is to prove that the rewriting relation generated by the rewrite rules together with β-reduction is confluent. This entails a property called product compatibility (also known as Π-injectivity or injectivity of function types) which, in turn, implies both subject reduction and uniqueness of types. Another important consequence of confluence is that, together with termination, it implies the decidability of the corresponding congruence. Indeed, for confluent and terminating relations, checking congruence boils down to a syntactic equality check between normal forms. As a direct corollary, we get the decidability of type checking in the $\lambda\Pi$-calculus Modulo for the corresponding rewrite relations.

One case where confluence is easily lost is if one allows rewrite rules with λ-abstractions on their left-hand side. For instance, consider the following rewrite rule (which reflects the mathematical equality $(e^f)' = f' * e^f$):

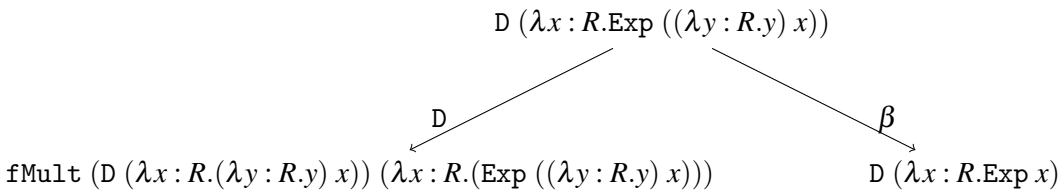
$$\mathtt{D}\ (\lambda x : R.\mathtt{Exp}\ (f\ x)) \hookrightarrow \mathtt{fMult}\ (\mathtt{D}\ (\lambda x : R.f\ x))\ (\lambda x : R.\mathtt{Exp}\ (f\ x)).$$

This rule introduces a non-joinable critical peak when combined with β-reduction:

Submitted to:
LFMTP 2015

x, y, z	$\in$	$\mathscr{V}$	(Variable)
$\underline{c}, \underline{f}$	$\in$	$\mathscr{C}_O$	(Object Constant)
C, F	$\in$	$\mathscr{C}_T$	(Type Constant)
$\underline{t}, \underline{u}, \underline{v}$	::=	$x \mid \underline{c} \mid \underline{u}\ \underline{v} \mid \lambda x : U.\underline{t}$	(Object)
U, V	::=	$C \mid U\ \underline{v} \mid \lambda x : U.V \mid \Pi x : U.V$	(Type)
K	::=	$\mathbf{Type} \mid \Pi x : U.K$	(Kind)
t, u, v	::=	$\underline{u} \mid U \mid K \mid \mathbf{Kind}$	(Term)

Figure 1: The terms of the $\lambda\Pi$-calculus Modulo



A way to recover confluence is to consider a generalized rewriting relation where matching is done modulo β-reduction. In this setting $\mathtt{D}\ (\lambda x : R.\mathtt{Exp}\ x)$ is reducible because it is β-equivalent to the redex $\mathtt{D}\ (\lambda x : R.\mathtt{Exp}((\lambda y : R.y)\ x))$ and, as we will see, this allows closing the critical peak.

In this paper, we formalize the notion of *rewriting modulo* β in the context of the $\lambda\Pi$-calculus Modulo. We achieve this by encoding the $\lambda\Pi$-calculus Modulo into Nipkow's Higher-Order Rewrite Systems [14]. This encoding allows us, first, to properly define matching modulo β using the notion of higher order rewriting and, secondly, to make available, in the $\lambda\Pi$-calculus Modulo, confluence and termination criteria designed for higher-order rewriting. Then we prove that the assumption of confluence for the rewriting modulo β relation can be used, in most proofs, in place of standard confluence. In particular this implies subject reduction (for both standard rewriting and rewriting modulo β) and uniqueness of types.

The paper is organized as follows. First, we define in Section 2 the $\lambda\Pi$-calculus modulo for which we prove subject reduction and uniqueness of types under the assumption of product compatibility and we show that confluence implies this latter property. In Section 3, we show that a naive definition of rewriting modulo β does not work in a typed setting. This leads us to use Higher-Order Rewrite Systems which we present in Section 4 and in which we encode the $\lambda\Pi$-calculus Modulo in Section 5. Then, we use this encoding to properly define rewriting modulo β in Section 6 and generalize the results of the previous sections. We discuss possible applications in Section 7 before concluding in Section 8.

2 The $\lambda\Pi$-Calculus Modulo

The $\lambda\Pi$-calculus Modulo is an extension of the dependently-typed λ-calculus ($\lambda\Pi$-calculus) where the β-conversion is extended by user-defined rewrite rules.

2.1 Terms

The terms of the $\lambda\Pi$-calculus Modulo are the same as the terms of the $\lambda\Pi$-calculus. Their syntax is given in Figure 1.

$$\begin{array}{lcll} \Delta & ::= & \emptyset \mid \Delta(x : U) & \text{(Local Context)} \\ \Gamma & ::= & \emptyset \mid \Gamma(\underline{c} : U) \mid \Gamma(C : K) \mid \Gamma(\underline{u} \hookrightarrow \underline{v}) \mid \Gamma(U \hookrightarrow V) & \text{(Global Context)} \end{array}$$

Figure 2: Syntax for contexts

Definition 2.1 (Object, Type, Kind, Term)**.** *A* term *is either an* object*, a* type*, a* kind *or the symbol* **Kind**.

An object is either a variable *in the set $\mathscr{V}$, or an* object constant *in the set $\mathscr{C}_O$, or an* application $\underline{u}\ \underline{v}$ *of two objects, or an* abstraction $\lambda x : A.\underline{t}$ *where A is a type and $\underline{t}$ is an object.*

A type is either a type constant *in the set $\mathscr{C}_T$, or an* application $U\ \underline{v}$ *where U is a type and $\underline{v}$ is an object, or an* abstraction $\lambda x : U.V$ *where U and V are types, or a* product $\Pi x : U.V$ *where U and V are types.*

A kind is either a product $\Pi x : U.K$ *where U is a type and K is a kind or the symbol* **Type**.

Type *and* **Kind** *are called* sorts.

The sets $\mathscr{V}$, $\mathscr{C}_O$ and $\mathscr{C}_T$ are assumed to be infinite and pairwise disjoint.

Definition 2.2. *A term is* algebraic *if it is not a variable, it is built from constants, variables and applications and variables do not have arguments.*

Notation 2.1. *In addition to the naming convention of Figure 1, we use A and B to denote types or kinds; T to denote a type, a kind or* **Kind***; s for* **Type** *or* **Kind**.

Moreover, we write $t\vec{u}$ to denote the application of t to an arbitrary number of arguments $u_1, \ldots, u_n$. We write $u[x/v]$ for the usual (capture-avoiding) substitution of x by v in u. We write $A \longrightarrow B$ for $\Pi x : A.B$ when B does not depend on x.

2.2 Contexts

We distinguish two kinds of context: local and global contexts. A local context is a list of typing declarations corresponding to variables. The syntax for contexts is given in Figure 2.

Definition 2.3 (Local Context)**.** *A* local context *is a list of variable declarations (variables together with their type).*

Following our previous work [17], we give a presentation of the $\lambda\Pi$-calculus Modulo where the rewrite rules are internalized in the system as part of the global context. This is a difference with earlier presentations [8] where the rewrite rules lived *outside* the system and were typed in an external system (either the simply-typed calculus or the $\lambda\Pi$-calculus). The main benefit of this approach is that the typing of the rewrite rules is made explicit and becomes an iterative process: rewrite rules previously added in the system can be used to type new ones.

Definition 2.4. *A* rewrite rule *is a pair of terms. We distinguish* object-level rewrite rules *(pairs of objects) from* type-level rewrite rules *(pairs of types).*

These are the only allowed rewrite rules. We write $(u \hookrightarrow v)$ for the rewrite rule (u, v).

It is left-algebraic *if u is algebraic and* left-linear *if no free variable occurs twice in u.*

Definition 2.5 (Global Context)**.** *A* global context *is a list of object declarations (an object constant together with a type), type declarations (a type constant together with a kind), object-level rewrite rules and type-level rewrite rules.*

$$\textbf{(Sort)}\quad \frac{}{\Gamma;\Delta \vdash \textbf{Type} : \textbf{Kind}}$$

$$\textbf{(Variable)}\quad \frac{(x:A)\in\Delta}{\Gamma;\Delta\vdash x:A}$$

$$\textbf{(Constant)}\quad \frac{(c:A)\in\Gamma}{\Gamma;\Delta\vdash c:A}$$

$$\textbf{(Application)}\quad \frac{\Gamma;\Delta\vdash t:\Pi x:A.B \qquad \Gamma;\Delta\vdash u:A}{\Gamma;\Delta\vdash tu:B[x/u]}$$

$$\textbf{(Abstraction)}\quad \frac{\Gamma;\Delta\vdash A:\textbf{Type} \qquad \Gamma;\Delta(x:A)\vdash t:B \qquad B\neq\textbf{Kind}}{\Gamma;\Delta\vdash \lambda x:A.t:\Pi x:A.B}$$

$$\textbf{(Product)}\quad \frac{\Gamma;\Delta\vdash A:\textbf{Type} \qquad \Gamma;\Delta(x:A)\vdash B:s}{\Gamma;\Delta\vdash \Pi x:A.B:s}$$

$$\textbf{(Conversion)}\quad \frac{\Gamma;\Delta\vdash t:A \qquad \Gamma;\Delta\vdash B:s \qquad A\equiv_{\beta\Gamma} B}{\Gamma;\Delta\vdash t:B}$$

Figure 3: Typing rules for terms in the $\lambda\Pi$-calculus Modulo.

2.3 Rewriting

Definition 2.6 (β-reduction)**.** *The* β-reduction *relation* $\to_\beta$ *is the smallest relation on terms containing* $(\lambda x:A.u)v \to_\beta u[x/v]$*, for all terms* A, u *and* v*, and closed by subterm rewriting.*

Definition 2.7 (Γ-reduction)**.** *Let* Γ *be a global context. The* Γ-reduction *relation* $\to_\Gamma$ *is the smallest relation on terms containing* $u \to_\Gamma v$ *for every rewrite rule* $(u \hookrightarrow v) \in \Gamma$*, closed by substitution and by subterm rewriting. We say that* $\to_\Gamma$ *is* left-algebraic *(respectively* left-linear*) if the rewrite rules in* Γ *are left-algebraic (respectively left-linear).*

Notation 2.2. *We write* $\to_{\beta\Gamma}$ *for* $\to_\beta \cup \to_\Gamma$*,* $\equiv_\beta$ *for the congruence generated by* $\to_\beta$ *and* $\equiv_{\beta\Gamma}$ *the congruence generated by* $\to_{\beta\Gamma}$*.*

It is important to notice that these notions of reduction are defined as relations on all (untyped) terms. In particular, we do not require the substitutions to be well-typed. This allows defining the notion of rewriting independently from the notion of typing (see below). This makes the system closer from what we would implement in practice.

Since the rewrite rules are either object-level or type-level, rewriting preserves the three syntactic categories (object, type, kind). Moreover, sorts are only convertible to themselves.

2.4 Type System

We now give the typing rules for the $\lambda\Pi$-calculus Modulo. We begin by the inference rules for terms, then for local contexts and finally for global contexts.

Definition 2.8 (Well-Typed Term)**.** *We say that a term* t has type A *in the global context* Γ *and the local context* Δ *if the judgment* $\Gamma;\Delta\vdash t:A$ *is derivable by the inference rules of Figure 3. We say that a term is* well-typed *if such* A *exists.*

$$\textbf{(Empty Local Context)} \qquad \frac{}{\Gamma \vdash^{ctx} \emptyset}$$

$$\textbf{(Variable Declaration)} \qquad \frac{\Gamma \vdash^{ctx} \Delta \qquad \Gamma;\Delta \vdash U : \textbf{Type} \qquad x \notin dom(\Delta)}{\Gamma \vdash^{ctx} \Delta(x : U)}$$

Figure 4: Typing rules for local contexts

The typing rules only differ from the usual typing rules for the $\lambda\Pi$-calculus by the **(Conversion)** rule where the congruence is extended from β-conversion to $\beta\Gamma$-conversion allowing taking into account the rewrite rules in the global context.

Definition 2.9 (Well-Formed Local Context)**.** *A local context Δ is* well-formed *with respect to a global context Γ if the judgment $\Gamma \vdash^{ctx} \Delta$ is derivable by the inference rules of Figure 4.*

Well-formed local contexts ensure that local declarations are unique and well-typed.

Besides the new conversion relation, the main difference between the $\lambda\Pi$-calculus and the $\lambda\Pi$-calculus Modulo is the presence of rewrite rules in global contexts. We need to take this into account when typing global contexts.

A key feature of any type system is the preservation of typing by reduction: the subject reduction property.

Definition 2.10 (Subject Reduction)**.** *Let Γ be a global context. We say that a rewriting relation $\rightarrow$ satisfies the* subject reduction *property in Γ if, for all terms t_1, t_2, T and local context Δ such that $\Gamma \vdash^{ctx} \Delta$, $\Gamma;\Delta \vdash t_1 : T$ and $t_1 \rightarrow t_2$ imply $\Gamma;\Delta \vdash t_2 : T$.*

In the $\lambda\Pi$-calculus Modulo, we cannot allow adding arbitrary rewrite rules in the context, if we want to preserve subject reduction. In particular, to prove subject reduction for the β-reduction we need the following property:

Definition 2.11 (Product-Compatibility)**.** *We say that a global context Γ satisfies the* product compatibility *property (and we note $\mathbf{PC}(\Gamma)$) if the following proposition is verified:*
if $\Pi x : A_1.B_1$ and $\Pi x : A_2.B_2$ are two well-typed product types in the same well-formed local context such that $\Pi x : A_1.B_1 \equiv_{\beta\Gamma} \Pi x : A_2.B_2$ then $A_1 \equiv_{\beta\Gamma} A_2$ and $B_1 \equiv_{\beta\Gamma} B_2$.

On the other hand, subject reduction for the Γ-reduction requires rewrite rules to be well-typed in the following sense:

Definition 2.12 (Well-typed Rewrite Rules)**.**

- *A rewrite rule $(u \hookrightarrow v)$ is* well-typed *for a global context Γ if, for any substitution σ, well-formed local context Δ and term T, $\Gamma;\Delta \vdash \sigma(u) : T$ implies $\Gamma;\Delta \vdash \sigma(v) : T$.*
- *A rewrite rule is* permanently well-typed *for a global context Γ if it is well-typed for any extension $\Gamma_0 \supset \Gamma$ that satisfies product compatibility. We write $\Gamma \vdash u \hookrightarrow v$ when $(u \hookrightarrow v)$ is permanently well-typed in Γ.*

The notion of permanently well-typed rewrite rule makes possible to typecheck rewrite rules only once and not each time we make new declarations or add other rewrite rules in the context.

We can now give the typing rules for global contexts.

Definition 2.13 (Well-formed Global Context)**.** *A global context is* well-formed *if the judgment Γ **wf** is derivable by the inference rules of Figure 5.*

(Empty Global Context)
$$\frac{}{\emptyset \textbf{ wf}}$$

(Object Declaration)
$$\frac{\Gamma \textbf{ wf} \qquad \Gamma;\emptyset \vdash U : \textbf{Type} \qquad \underline{c} \notin dom(\Gamma)}{\Gamma(\underline{c} : U) \textbf{ wf}}$$

(Type Declaration)
$$\frac{\Gamma \textbf{ wf} \qquad \Gamma;\emptyset \vdash K : \textbf{Kind} \qquad \textbf{PC}(\Gamma(C : K)) \qquad C \notin dom(\Gamma)}{\Gamma(C : K) \textbf{ wf}}$$

(Rewrite Rules)
$$\frac{\Gamma \textbf{ wf} \qquad (\forall i)\Gamma \vdash u_i \hookrightarrow v_i \qquad \textbf{PC}(\Gamma(u_1 \hookrightarrow v_1)\dots(u_n \hookrightarrow v_n))}{\Gamma(u_1 \hookrightarrow v_1)\dots(u_n \hookrightarrow v_n) \textbf{ wf}}$$

Figure 5: Typing rules for global contexts

The rules **(Object Declaration)** and **(Type Declaration)** ensure that constant declarations are well-typed. One can remark that the premise $\textbf{PC}(\Gamma(\underline{c} : U))$ is *missing* in the **(Object Declaration)** rule. This is because $\textbf{PC}(\Gamma(\underline{c} : U))$ can be proved from $\textbf{PC}(\Gamma)$; to prove product compatibility for $\Gamma(c : U)$ it suffices to emulate the constant c by a fresh variable and use the product compatibility property of Γ. This cannot be done for type declarations since type-level variables do not exist in the $\lambda\Pi$-calculus Modulo. The rule **(Rewrite Rules)** permits adding rewrite rules. Notice that we can add several rewrite rules at once. In this case, only product compatibility for the whole system is required. On the other hand, when a rewrite rule is added it needs to be well-typed independently from the other rules that are added at the same time.

Well-formed global contexts satisfy subject reduction and uniqueness of types. Proofs can be found in the long version of this paper at the author's webpage.

Theorem 2.1 (Subject Reduction)**.** *Let Γ be a well-formed global context. Subject reduction holds for $\rightarrow_{\beta\Gamma}$ in Γ.*

Theorem 2.2 (Uniqueness of Types)**.** *Let Γ be a well-formed global context and let Δ be a local context well-formed for Γ. If $\Gamma;\Delta \vdash t : T_1$ and $\Gamma;\Delta \vdash t : T_2$ then $T_1 \equiv_{\beta\Gamma} T_2$.*

Remark that strong normalization of well-typed terms for the relations $\rightarrow_\Gamma$ and $\rightarrow_\beta$ is not guaranteed.

2.5 Criteria for Product Compatibility and Well-typedness of Rewrite Rules

We now give effective criteria for checking product compatibility and well-typedness of rewrite rules.

The usual way to prove product compatibility is by showing the confluence of the rewrite system.

Theorem 2.3 (Product Compatibility from Confluence)**.** *Let Γ be a global context. If $\rightarrow_{\beta\Gamma}$ is confluent then product compatibility holds for Γ.*

One could think that we can weaken the assumption of confluence requiring only confluence for well-typed terms. This is not a viable option since, without product compatibility, we do not know if reduction preserves typing (subject reduction) and if the set of well-typed terms is closed by reduction. Therefore, it seems unlikely to be able to prove confluence only for well-typed terms before proving the product compatibility property.

The confluence of $\rightarrow_{\beta\Gamma}$ can be obtained from the confluence of $\rightarrow_\Gamma$.

Theorem 2.4 (Müller [12])**.** *If $\rightarrow_\Gamma$ is left-algebraic, left-linear and confluent, then $\rightarrow_{\beta\Gamma}$ is confluent.*

To show that a rewrite rule is well-typed, one can use the following result:

Theorem 2.5. *Let Γ be a well-formed global context and $(u \hookrightarrow v)$ be a rewrite rule. If u is algebraic and there exist Δ and T such that $\Gamma \vdash^{ctx} \Delta$, $dom(\Delta) = FV(u)$, $\Gamma;\Delta \vdash u : T$ and $\Gamma;\Delta \vdash v : T$ then $(u \hookrightarrow v)$ is permanently well-typed for Γ.*

2.6 Example

As an example, we define the map function on lists of integers. We first define the type of *Peano integers* by the three successive global declarations:

$\texttt{Nat} : \textbf{Type}.$
$0 : \texttt{Nat}.$
$\texttt{S} : \texttt{Nat} \longrightarrow \texttt{Nat}.$

For readability, we will write n instead of $\overbrace{S\ (S\ \ldots (S}^{n\ times}\ 0))$. We now define a type for lists:

$\texttt{List} : \textbf{Type}.$
$\texttt{Nil} : \texttt{List}.$
$\texttt{Cons} : \texttt{Nat} \longrightarrow \texttt{List} \longrightarrow \texttt{List}.$

and the function map on lists:

$\texttt{Map} : (\texttt{Nat} \longrightarrow \texttt{Nat}) \longrightarrow \texttt{List} \longrightarrow \texttt{List}.$
$\texttt{Map}\ f\ \texttt{Nil} \hookrightarrow \texttt{Nil}.$
$\texttt{Map}\ f\ (\texttt{Cons}\ hd\ tl) \hookrightarrow \texttt{Cons}\ (f\ hd)\ (\texttt{Map}\ f\ tl).$

For instance, we can use this function to add some value to the elements of a list. First, we define addition:

$\texttt{plus} : \texttt{Nat} \longrightarrow \texttt{Nat} \longrightarrow \texttt{Nat}.$
$\texttt{plus}\ 0\ n \hookrightarrow n.$
$\texttt{plus}\ (\texttt{S}\ n_1)\ n_2 \hookrightarrow \texttt{S}\ (\texttt{plus}\ n_1\ n_2).$

Then, we have the following reduction:

$\texttt{Map}\ (\texttt{plus}\ 3)\ (\texttt{Cons}\ 1\ (\texttt{Cons}\ 2\ (\texttt{Cons}\ 3\ \texttt{Nil}))) \to_\Gamma^* \texttt{Cons}\ 4\ (\texttt{Cons}\ 5\ (\texttt{Cons}\ 6\ \texttt{Nil}))$.

This global context is well-formed. Indeed, one can check that each global declaration is well-typed. Moreover, each time we add a rewrite rule, it verifies the hypotheses of Theorem 2.5 and it preserves the confluence of the relation $\to_{\beta\Gamma}$. Therefore, the rewrite rules are permanently well-typed and, by Theorem 2.3, product compatibility is always guaranteed.

3 A Naive Definition of Rewriting Modulo β

As already mentioned, our goal is to give a notion of rewriting modulo β in the setting of $\lambda\Pi$-calculus Modulo. We first exhibit the issues arising from a naive definition of this notion.

In an untyped setting, we could define rewriting modulo β in this manner: t_1 rewrites to t_2 if, for some rewrite rule $(u \hookrightarrow v)$ and substitution σ, $\sigma(u) \equiv_\beta t_1$ and $\sigma(v) \equiv_\beta t_2$. This definition is not satisfactory for several reasons.

It breaks subject reduction. For the rewrite rule of Section 1, taking $\sigma = \{f \mapsto \lambda y : \Omega.y\}$ where Ω is some ill-typed term, we have

$$\texttt{D}\ (\lambda x : R.\texttt{Exp}\ x) \longrightarrow \texttt{fMult}\ (\texttt{D}\ (\lambda x : R.(\lambda y : \Omega.y)\ x)\ (\lambda x : R.\texttt{Exp}\ ((\lambda y : \Omega.y)\ x)))$$

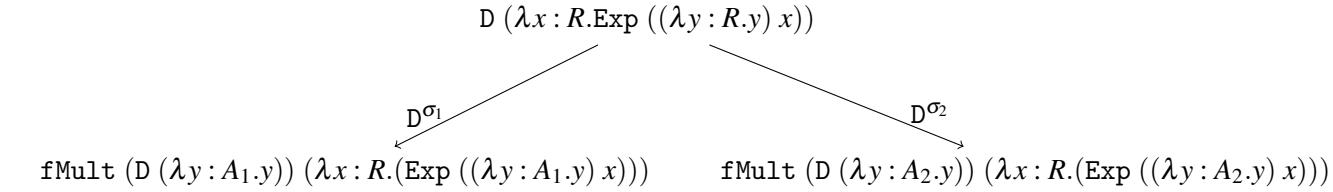
and, even if $\texttt{D}\ (\lambda x : R.\texttt{Exp}\ x)$ is well-typed, its reduct is ill-typed since it contains an ill-typed subterm.

It may introduce free variables. In the example above, Ω has no reason to be closed.

It does not provide confluence. If we consider the following variant of the rewrite rule

$$\mathtt{D}\ (\lambda x : R.\mathtt{Exp}\ (f\ x)) \hookrightarrow \mathtt{fMult}\ (\mathtt{D}\ f)\ (\lambda x : R.\mathtt{Exp}\ (f\ x))$$

and take $\sigma_1 = \{f \mapsto \lambda y : A_1.y\}$ and $\sigma_2 = \{f \mapsto \lambda y : A_2.y\}$ where A_1 and A_2 are two non convertible types then we have:



and the peak is not joinable.

Therefore, we need to find a definition that takes care of these issues. We will achieve this using an embedding of $\lambda\Pi$-calculus Modulo into Higher-Order Rewrite Systems.

4 Higher-Order Rewrite Systems

In 1991, Nipkow [14] introduced Higher-Order Rewrite Systems (HRS) in order to lift termination and confluence results from first-order rewriting to rewriting over λ-terms. More generally, the goal was to study rewriting over terms with bound variables such as programs, theorem and proofs.

Unlike the $\lambda\Pi$-calculus Modulo, in HRSs β-reduction and rewriting do not operate at the same level. Rewriting is defined as a relation between the $\beta\eta$-equivalence classes of simply typed λ-terms: the λ-calculus is used as a meta-language.

Higher-Order Rewrite Systems are based upon the (pre)terms of the simply-typed λ-calculus built from a signature. A signature is a set of base types $\mathscr{B}$ and a set of typed constants. A simple type is either a base type $b \in \mathscr{B}$ or an arrow $A \longrightarrow B$ where A and B are simple types.

Definition 4.1 (Preterm)**.** *A* preterm *of type A is*

- *either a* variable *x of type A (we assume given for each simple type A an infinite number of variables of this type),*
- *or a* constant *f of type A,*
- *or an* application *$t(u)$ where t is a preterm of type $B \longrightarrow A$ and u is a preterm of type B,*
- *or, if $A = B \longrightarrow C$, an* abstraction *$\underline{\lambda}x.t$ where x is a variable of type B and t is a preterm of type C.*

In order to distinguish the abstraction of HRSs from the abstraction of $\lambda\Pi$-calculus Modulo, we use the underlined symbol $\underline{\lambda}$ instead of λ. Similarly, we write the application $t(u)$ for HRSs (instead of tu). We use the abbreviation $t(u_1,\ldots,u_n)$ for $t(u_1)\ldots(u_n)$. If A is a simple type, we write A^1 for A and A^{n+1} for $A \longrightarrow A^n$.

Notice also that HRSs abstractions do not have type annotations because variables are typed.

β-reduction and η-expansion are defined as usual on preterms. We write $\updownarrow^{\eta}_{\beta} t$ for the long $\beta\eta$-normal form of t.

Definition 4.2 (Term)**.** *A* term *is a preterm in long $\beta\eta$-normal form.*

Definition 4.3 (Pattern)**.** *A term t is a* pattern *if every free occurrence of a variable F is in a subterm of t of the form $F\vec{u}$ such that $\vec{u}$ is η-equivalent to a list of distinct bound variables.*

The crucial result about patterns (due to Miller [11]) is the decidability of higher-order unification (unification modulo $\beta\eta$) of patterns. Moreover, if two patterns are unifiable then a most general unifier exists and is computable.

The notion of rewrite rule for HRSs is the following:

Definition 4.4 (Rewrite Rules)**.** *A* rewrite rule *is a pair of terms $(l \hookrightarrow r)$ such that l is a pattern not η-equivalent to a variable, $FV(r) \subset FV(l)$ and l and r have the same base type.*

The restriction to patterns for the left-hand side ensures that matching is decidable but also that, when it exists, the resulting substitution is unique. This way, the situation is very close to first-order (*i.e.* syntactic) matching.

Definition 4.5 (Higher-Order Rewriting System (HRS))**.** *A* Higher-Order Rewriting System *is a set R of rewrite rules.*

The rewrite relation $\to_R$ is the smallest relation on terms closed by subterm rewriting such that, for any $(l \hookrightarrow r) \in R$ and any well-typed substitution σ, $\updownarrow_\beta^\eta \sigma(l) \to_R \updownarrow_\beta^\eta \sigma(r)$.

The standard example of an HRS is the untyped λ-calculus. The signature involves a single base type `Term` and two constants:

$$\texttt{Lam} : (\texttt{Term} \longrightarrow \texttt{Term}) \longrightarrow \texttt{Term}$$

$$\texttt{App} : \texttt{Term} \longrightarrow \texttt{Term} \longrightarrow \texttt{Term}$$

and a single rewrite rule for β-reduction:

$$(beta) \quad \texttt{App}(\texttt{Lam}(\underline{\lambda} x.X(x)), Y) \hookrightarrow X(Y)$$

5 An Encoding of the $\lambda\Pi$-calculus Modulo into Higher-Order Rewrite Systems

5.1 Encoding of Terms

We now mimic the encoding of the untyped λ-calculus as an HRS and encode the terms of the $\lambda\Pi$-calculus Modulo. First we specify the signature.

Definition 5.1. *The signature* $\mathbf{Sig}(\lambda\Pi)$ *is composed of a single base type* `Term`*, the constants* `Type` *and* `Kind` *of atomic type* `Term`*, the constant* `App` *of type* $\texttt{Term} \longrightarrow \texttt{Term} \longrightarrow \texttt{Term}$*, the constants* `Lam` *and* `Pi` *of type* $\texttt{Term} \longrightarrow (\texttt{Term} \longrightarrow \texttt{Term}) \longrightarrow \texttt{Term}$ *and the constants* `c` *of type* `Term` *for every constant $c \in \mathscr{C}_O \cup \mathscr{C}_T$.*

Then we define the encoding of $\lambda\Pi$-terms.

Definition 5.2 (Encoding of $\lambda\Pi$-term)**.** *The function $\|.\|$ from $\lambda\Pi$-terms to HRS-terms in the signature* $\mathbf{Sig}(\lambda\Pi)$ *is defined as follows:*

$$\begin{array}{llll} \|\mathbf{Kind}\| & := & \texttt{Kind} & \|\mathbf{Type}\| := \texttt{Type} \\ \|x\| & := & x \text{ (variable of type } \texttt{Term}) & \|c\| := \texttt{c} \\ \|uv\| & := & \texttt{App}(\|u\|, \|v\|) & \|\lambda x : A.t\| := \texttt{Lam}(\|A\|, \underline{\lambda} x.\|t\|) \\ \|\Pi x : A.B\| & := & \texttt{Pi}(\|A\|, \underline{\lambda} x.\|B\|) & \end{array}$$

Lemma 5.1. *The function $\|.\|$ is a bijection from the $\lambda\Pi$-terms to HRS-terms of type* `Term`*.*

Note that this is a bijection between the untyped terms of the $\lambda\Pi$-calculus Modulo and well-typed terms of the corresponding HRS.

5.2 Higher-Order Rewrite Rules

We have faithfully encoded the terms. The next step is to encode the rewrite rules. The following rule corresponds to β-reduction at the HRS level:

$$(beta) \quad \mathtt{App}(\mathtt{Lam}(X, \underline{\lambda} x.Y(x)), Z) \hookrightarrow Y(Z)$$

We have the following correspondence:

Lemma 5.2.

- *If* $t_1 \to_\beta t_2$ *then* $\|t_1\| \to_{(beta)} \|t_2\|$.
- *If* $t_1 \to_{(beta)} t_2$ *and* t_1, t_2 *have type* `Term` *then* $\|t_1\|^{-1} \to_\beta \|t_2\|^{-1}$ *(where* $\|.\|^{-1}$ *is the inverse of* $\|.\|$*).*

By encoding rewrite rules in the obvious way (translating $(u \hookrightarrow v)$ by $(\|u\| \hookrightarrow \|v\|)$), we would get a similar result for Γ-reduction. But, since we want to incorporate rewriting modulo β, we proceed differently.

First, we introduce the notion of uniform terms. These are terms verifying an arity constraint on their free variables.

Definition 5.3 (Uniform Terms)**.** *A term* t *is* uniform *for a set of variables* V *if all occurrences of a variable free in* t *not in* V *is applied to the same number of arguments.*

Now, we define an encoding for uniform terms.

Definition 5.4 (Encoding of uniform terms)**.** *Let* V *be a set of variables and* t *be a term uniform in* V*. The HRS-term* $\|u\|_V$ *of type* `Term` *is defined as follows:*

$\|\mathbf{Kind}\|_V$	:=	`Kind`
$\|\mathbf{Type}\|_V$	:=	`Type`
$\|x\|_V$	:=	x if $x \in V$ (variable of type `Term`)
$\|c\|_V$	:=	`c`
$\|\lambda x : A.u\|_V$	:=	$\mathtt{Lam}(\|A\|_V, \underline{\lambda} x.\|u\|_{V \cup \{x\}})$
$\|\Pi x : A.B\|_V$	:=	$\mathtt{Pi}(\|A\|_V, \underline{\lambda} x.\|B\|_{V \cup \{x\}})$
$\|x\vec{v}\|_V$	:=	$x(\|\vec{v}\|_V)$ if $x \notin V$ (x of type $\mathtt{Term}^{n+1}$ where $n = \|\vec{v}\|$)
$\|uv\|_V$	:=	$\mathtt{App}(\|u\|_V, \|v\|_V)$ if $uv \neq x\ \vec{w}$ for $x \notin V$

Now, we define an equivalent of patterns for the $\lambda\Pi$-calculus Modulo.

Definition 5.5 ($\lambda\Pi$-patterns)**.** *Let* V_0 *be a set of variables,* $\mathscr{A}$ *be a function giving an arity to variables and let* $V = (V_0, \mathscr{A})$*. The subset* $\mathscr{P}_V$ *of* $\lambda\Pi$*-terms is defined inductively as follows:*

- *if* c *is a constant, then* $c \in \mathscr{P}_V$*;*
- *if* $p, q \in \mathscr{P}_V$*, then* $p\ q \in \mathscr{P}_V$*;*
- *if* $x \in V_0$*, then* $x \in \mathscr{P}_V$*;*
- *if* $p \in \mathscr{P}_V$*,* $x \notin V_0$ *and* $\vec{y}$ *is a vector of pairwise distinct variables in* V_0 *such that* $|\vec{y}| = \mathscr{A}(x)$*, then* $p\ (x\ \vec{y}) \in \mathscr{P}_V$*;*
- *if* $p \in \mathscr{P}_V$*,* $FV(A) \subset V_0$ *and* $q \in \mathscr{P}_{(V_0 \cup \{x\}, \mathscr{A})}$*, then* $p\ (\lambda x : A.q) \in \mathscr{P}_V$*;*

A term t *is a* $\lambda\Pi$-pattern *if, for some arity function* $\mathscr{A}$*,* $t \in \mathscr{P}_{(\emptyset, \mathscr{A})}$.

Remark that the encoding of a $\lambda\Pi$-pattern as a uniform term is a pattern.

We now define the encoding of rewrite rules.

Definition 5.6 (Encoding of Rewrite Rules)**.** *Let $(u \hookrightarrow v)$ be a rewrite rule such that*

- *u is a $\lambda\Pi$-pattern;*
- *$FV(v) \subset FV(u)$;*
- *all free occurrences of a variable in u and v are applied to the same number of arguments.*

The encoding of $(u \hookrightarrow v)$ is $\|u \hookrightarrow v\| = \|u\|_\emptyset \hookrightarrow \|v\|_\emptyset$.

Remark that the first assumption ensures that the left-hand side is a pattern and the third assumption ensures that the HRS-term is well-typed.

Definition 5.7 (HRS(Γ))**.** *Let Γ a global context whose rewrite rules satisfy the condition of Definition 5.6. We write HRS(Γ) for the HRS $\{\|u \hookrightarrow v\| \mid (u \hookrightarrow v) \in \Gamma\}$ and HRS($\beta\Gamma$) for HRS$(\Gamma) \cup \{(beta)\}$.*

6 Rewriting Modulo β

6.1 Definition

We are now able to properly define rewriting modulo β. As for usual rewriting, rewriting modulo β is defined on all (untyped) terms.

Definition 6.1 (Rewriting Modulo β)**.** *Let Γ be a global context. We say that t_1* rewrites to *t_2* modulo β *(written $t_1 \to_{\Gamma^b} t_2$) if $\|t_1\|$ rewrites to $\|t_2\|$ in HRS(Γ). Similarly, we write $t_1 \to_{\beta\Gamma^b} t_2$ if $\|t_1\|$ rewrites to $\|t_2\|$ in HRS($\beta\Gamma$).*

Lemma 6.1.

- $\to_{\beta\Gamma^b} = \to_{\Gamma^b} \cup \to_\beta$.
- *If $t_1 \to_\Gamma t_2$ then $t_1 \to_{\Gamma^b} t_2$.*

6.2 Example

Let us look at the example from the introduction. Now we have :

$$\mathtt{D}\ (\lambda x : R.\mathtt{Exp}\ x) \to_{\Gamma^b} \mathtt{fMult}\ (\mathtt{D}\ (\lambda x : R.x))\ (\lambda x : R.\mathtt{Exp}\ x)$$

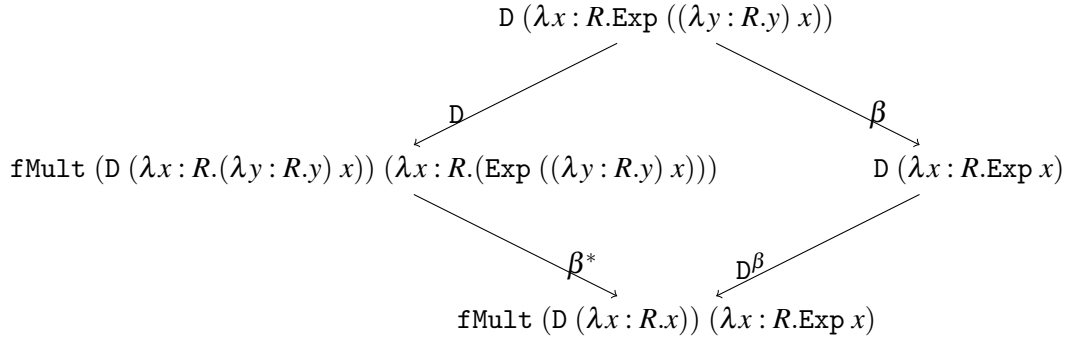
Indeed, for $\sigma = \{f \mapsto \underline{\lambda} y.y\}$ we have

$$\|\mathtt{D}\ (\lambda x : R.\mathtt{Exp}\ x)\| = \mathtt{App}(\mathtt{D}, \mathtt{Lam}(R, \underline{\lambda} x.\mathtt{App}(\mathtt{Exp}, x))) =^{\updownarrow^\eta}_\beta \sigma(\mathtt{App}(\mathtt{D}, \mathtt{Lam}(R, \underline{\lambda} x.\mathtt{App}(\mathtt{Exp}, f(x)))))$$

and

$$\begin{aligned}\|\mathtt{fMult}\ (\mathtt{D}\ (\lambda x : R.x))\ (\lambda x : R.\mathtt{Exp}\ x)\| &= \mathtt{App}(\mathtt{fMult}, \mathtt{App}(\mathtt{D}, \mathtt{Lam}(R, \underline{\lambda} x.x)), \mathtt{Lam}(R, \underline{\lambda} x.\mathtt{App}(\mathtt{Exp}, x))) \\ &=^{\updownarrow^\eta}_\beta \sigma(\mathtt{App}(\mathtt{fMult}, \mathtt{App}(\mathtt{D}, \mathtt{Lam}(R, \underline{\lambda} x.f(x))), \mathtt{Lam}(R, \underline{\lambda} x.\mathtt{App}(\mathtt{Exp}, f(x)))))\end{aligned}$$

Therefore, the peak is now joinable.



In fact the rewriting relation can be shown confluent [15].

6.3 Properties

Rewriting modulo β also preserves typing.

Theorem 6.1 (Subject Reduction for $\rightarrow_{\Gamma^b}$)**.** *Let Γ a well-formed global context and Δ a local context well-formed for Γ. If $\Gamma;\Delta \vdash t_1 : T$ and $t_1 \rightarrow_{\Gamma^b} t_2$ then $\Gamma;\Delta \vdash t_2 : T$.*

It directly follows from the following lemma:

Lemma 6.2. *If $t_1 \rightarrow_{\Gamma^b} t_2$ then, for some t_1' and t_2', we have $t_1 \leftarrow_\beta^* t_1' \rightarrow_\Gamma t_2' \rightarrow_\beta^* t_2$. Moreover, if t_1 is well-typed then we can choose t_1' such that it is well-typed in the same context.*

Proof. The idea is to lift the β-reductions that occur at the HRS level to the $\lambda\Pi$-calculus Modulo. Suppose $t_1 \rightarrow_{\Gamma^b} t_2$. For some rewrite rule $(u \hookrightarrow v)$ and (HRS) substitution σ, we have $\updownarrow_\beta^\eta \sigma(u) = \|t_1\|$ and $\updownarrow_\beta^\eta \sigma(v) = \|t_2\|$. We define the ($\lambda\Pi$) substitution $\hat{\sigma}$ as follows: $\hat{\sigma}(x) = \|\sigma(x)\|^{-1}$ if $\sigma(x)$ has type $\mathtt{Term}$; $\hat{\sigma}(x) = \lambda\vec{x} : \vec{A}.\|u\|^{-1}$ if $\sigma(x) = \underline{\lambda}\vec{x}.u$ has type $\mathtt{Term}^n \longrightarrow \mathtt{Term}$ where the A_i are arbitrary types. We have, at the $\lambda\Pi$ level, $\hat{\sigma}(u) \rightarrow_\Gamma \hat{\sigma}(v)$, $\hat{\sigma}(u) \rightarrow_\beta^* t_1$ and $\hat{\sigma}(v) \rightarrow_\beta^* t_2$. If t_1 is well-typed then the A_i can be chosen so that $\hat{\sigma}(u)$ is also well-typed. □

Another consequence of this lemma is that the rewriting modulo β does not modify the congruence.

Theorem 6.2. *The congruence generated by $\rightarrow_{\beta\Gamma^b}$ is equal to $\equiv_{\beta\Gamma}$.*

Proof. Follows from Lemma 6.1 and Lemma 6.2. □

6.4 Generalized Criteria for Product Compatibility and Well-Typedness of Rewrite Rules

Using our new notion of rewriting modulo β, we can generalize the criteria of Section 2.5.

Theorem 6.3. *Let Γ be a global context. If HRS($\beta\Gamma$) is confluent, then product compatibility holds for Γ.*

Proof. Assume that $\Pi x : A_1.B_1 \equiv_{\beta\Gamma} \Pi x : A_2.B_2$ then, by Theorem 6.2, $\Pi x : A_1.B_1 \equiv_{\beta\Gamma^b} \Pi x : A_2.B_2$. By confluence, there exist A_0 and B_0 such that $A_1 \rightarrow_{\beta\Gamma^b}^* A_0$, $A_2 \rightarrow_{\beta\Gamma^b}^* A_0$, $B_1 \rightarrow_{\beta\Gamma^b}^* B_0$ and $B_2 \rightarrow_{\beta\Gamma^b}^* B_0$. It follows, by Theorem 6.2, that $A_1 \equiv_{\beta\Gamma} A_2$ and $B_1 \equiv_{\beta\Gamma} B_2$. □

To prove the confluence of a HRS, one can use van Oostrom's development-closed theorem [15].

Theorem 2.5 can also be generalized to deal with $\lambda\Pi$-patterns.

Theorem 6.4. *Let Γ be a well-formed global context and $(u \hookrightarrow v)$ be a rewrite rule. If u is a $\lambda\Pi$-pattern and there exist Δ and T such that $\Gamma \vdash^{ctx} \Delta$, $FV(u) = dom(\Delta)$, $\Gamma;\Delta \vdash u : T$ and $\Gamma;\Delta \vdash v : T$ then $(u \hookrightarrow v)$ is permanently well-typed for Γ.*

This theorem is a corollary of the following lemma.

Lemma 6.3. *Let $\Gamma \subset \Gamma_2$ be two well-formed global contexts. If $t \in \mathscr{P}_{dom(\Sigma)}$, $dom(\sigma) = dom(\Delta)$, for all $(x : A) \in \Sigma$, $\sigma(A) = A$, $\Gamma;\Delta\Sigma \vdash t : T$ and $\Gamma_2;\Delta_2\Sigma \vdash \sigma(t) : T_2$ then $T_2 \equiv_{\beta\Gamma_2} \sigma(T)$ and, for all $x \in FV(t) \cap dom(\Delta)$, $\Gamma_2;\Delta_2 \vdash \sigma(x) : T_x$ for $T_x \equiv_{\beta\Gamma_2} \sigma(\Delta(x))$.*

Proof. We proceed by induction on $t \in \mathscr{P}_{dom(\Sigma)}$.

- if $t = c$ is a constant, then $FV(t) = \emptyset$ and, by inversion on $\Gamma;\Delta\Sigma \vdash t : T$, there exists a (closed term) A such that $(c : A) \in \Gamma \subset \Gamma_2$, $T \equiv_{\beta\Gamma} A$ and $T_2 \equiv_{\beta\Gamma_2} A$. Since $A = \sigma(A)$, we have $\sigma(T) \equiv_{\beta\Gamma_2} T_2$.

- if $t = x \in dom(\Sigma)$, then, by inversion, there exists A such that $(x : A) \in \Sigma$, $T \equiv_{\beta\Gamma} A$ and $T_2 \equiv_{\beta\Gamma_2} A$. Since $A = \sigma(A)$, we have $\sigma(T) \equiv_{\beta\Gamma_2} T_2$.
- if $t = p\ q$, then, by inversion, on the one hand, $\Gamma;\Delta\Sigma \vdash p : \Pi x : A.B$, $\Gamma;\Delta\Sigma \vdash q : A$ and $T \equiv_{\beta\Gamma} B[x/q]$. On the other hand, $\Gamma_2;\Delta_2\Sigma \vdash \sigma(p) : \Pi x : A_2.B_2$, $\Gamma_2;\Delta_2\Sigma \vdash \sigma(q) : A_2$ and $T_2 \equiv_{\beta\Gamma_2} B_2[x/\sigma(q)]$.
  By induction hypothesis on p, we have $\sigma(\Pi x : A.B) \equiv_{\beta\Gamma_2} \Pi x : A_2.B_2$ and for all $x \in FV(p) \cap dom(\Delta)$, $\Gamma_2;\Delta_2 \vdash \sigma(x) : T_x$ with $T_x \equiv_{\beta\Gamma_2} \sigma(\Delta(x))$.
  By product-compatibility of Γ_2, $\sigma(A) \equiv_{\beta\Gamma_2} A_2$ and $\sigma(B) \equiv_{\beta\Gamma_2} B_2$. It follows that $\sigma(T) \equiv_{\beta\Gamma_2} \sigma(B[x/q]) \equiv_{\beta\Gamma_2} B_2[x/\sigma(q)] \equiv_{\beta\Gamma_2} T_2$.
  Now, we distinguish three sub-cases:
  - either $q \in \mathscr{P}_{dom(\Sigma)}$ and by induction hypothesis on q, for all $x \in FV(q) \cap dom(\Delta)$, $\Gamma_2;\Delta_2 \vdash \sigma(x) : T_x$ with $T_x \equiv_{\beta\Gamma_2} \sigma(\Delta(x))$.
  - Or $q = \lambda x : A.q_0$ with $FV(A) \in dom(\Sigma)$ and $q_0 \in \mathscr{P}_{dom(\Sigma(x:A))}$ and by induction hypothesis on q_0, for all $x \in FV(q_0) \cap dom(\Delta)$, $\Gamma_2;\Delta_2 \vdash \sigma(x) : T_x$ with $T_x \equiv_{\beta\Gamma_2} \sigma(\Delta(x))$.
  - Or $q = x\vec{y}$ with $x \notin dom(\Sigma)$ and $\vec{y} \subset dom(\Sigma)$. By inversion, on the one hand, $\Delta(x) \equiv_{\beta\Gamma} \Pi\vec{y} : \Sigma(\vec{y}).C$ for $C \equiv_{\beta\Gamma} A$. On the other hand, $\Gamma_2;\Delta_2 \vdash \sigma(x) : \Pi\vec{y} : \Sigma(\vec{y}).C_2$ for $C_2 \equiv_{\beta\Gamma_2} A_2$. Since $\sigma(A) \equiv_{\beta\Gamma_2} A_2$, we have $\Pi\vec{y} : \Sigma(\vec{y}).C_2 \equiv_{\beta\Gamma_2} \Pi\vec{y} : \Sigma(\vec{y}).\sigma(C) = \sigma(\Delta(x))$.

□

Proof of Theorem 6.4. Let Γ_2 be a well-formed extension of Γ. Suppose that $\Gamma_2;\Delta_2 \vdash \sigma(u) : T_2$.

By Lemma 6.3 and $FV(u) = dom(\Delta)$, we have, for all $x \in dom(\Delta)$, $\Gamma_2;\Delta_2 \vdash \sigma(x) : T_x$ for $T_x \equiv_{\beta\Gamma_2} \sigma(\Delta(x))$ and $T_2 \equiv_{\beta\Gamma_2} \sigma(T)$.

By induction on $\Gamma;\Delta \vdash v : T$, we deduce $\Gamma_2;\Delta_2 \vdash \sigma(v) : T_3$, for $T_3 \equiv_{\beta\Gamma_2} \sigma(T) \equiv_{\beta\Gamma_2} T_2$. It follows, by conversion, that $\Gamma_2;\Delta_2 \vdash \sigma(v) : T_2$. □

7 Applications

7.1 Parsing and Solving Equations

The context declarations and rewrite rules of Figure 6 define a function `to_expr` which parses a function of type `Nat` to `Nat` into an expression of the form $a * x + b$ (represented by the term `mk_expr` a b) where a and b are constants. The left-hand sides of the rewrite rules on `to_expr` are $\lambda\Pi$-patterns. This allows defining `to_expr` by pattern matching in a way which looks under the binders.

The function `solve` can then be used to solve the linear equation $a * x + b = 0$. The answer is either `None` if there is no solution, or `All` if any x is a solution or `One` m n if $-m/(n+1)$ is the only solution.

For instance, we have (writing `One` $-\frac{1}{3}$ for `One 1 2`):

$$\texttt{solve}\ (\texttt{to\_expr}(\lambda x : \texttt{Nat.plus}\ x\ (\texttt{plus}\ x\ (\texttt{S}\ x)))) \to^*_{\beta\Gamma} \texttt{One}\ -\frac{1}{3}.$$

By Theorem 6.3 and Theorem 6.4 the global context of Figure 6 is well-formed.

7.2 Universe Reflection

In [1], Assaf defines a version of the calculus of construction with explicit universe subtyping thanks to an extended notion of conversion generated by a set of rewrite rules. This work can easily be adapted to fit in the framework of the $\lambda\Pi$-calculus Modulo. However, the confluence of the rewrite system holds only for rewriting modulo β.

$$
\begin{array}{lcl}
\texttt{expr} & : & \textbf{Type}. \\
\texttt{mk\_expr} & : & \texttt{Nat} \longrightarrow \texttt{Nat} \longrightarrow \texttt{expr}. \\
\texttt{expr\_S} & : & \texttt{expr} \longrightarrow \texttt{expr}. \\
\texttt{expr\_S}\ (\texttt{mk\_expr}\ a\ b) & \hookrightarrow & \texttt{mk\_expr}\ a\ (\texttt{S}\ b). \\
\texttt{expr\_P} & : & \texttt{expr} \longrightarrow \texttt{expr} \longrightarrow \texttt{expr}. \\
\texttt{expr\_P}\ (\texttt{mk\_expr}\ a_1\ b_1)\ (\texttt{mk\_expr}\ a_2\ b_2) & \hookrightarrow & \texttt{mk\_expr}\ (\texttt{plus}\ a_1\ a_2)\ (\texttt{plus}\ b_1\ b_2). \\
\texttt{to\_expr} & : & (\texttt{Nat} \longrightarrow \texttt{Nat}) \longrightarrow \texttt{expr}. \\
\texttt{to\_expr}\ (\lambda x : \texttt{Nat}.0) & \hookrightarrow & \texttt{mk\_expr}\ 0\ 0. \\
\texttt{to\_expr}\ (\lambda x : \texttt{Nat}.\texttt{S}\ (f\ x)) & \hookrightarrow & \texttt{expr\_S}\ (\texttt{to\_expr}\ (\lambda x : \texttt{Nat}.f\ x)). \\
\texttt{to\_expr}\ (\lambda x : \texttt{Nat}.x) & \hookrightarrow & \texttt{mk\_expr}\ (\texttt{S}\ 0)\ 0. \\
\texttt{to\_expr}\ (\lambda x : \texttt{Nat}.\texttt{plus}\ (f\ x)\ (g\ x)) & \hookrightarrow & \\
\multicolumn{3}{r}{\texttt{expr\_P}\ (\texttt{to\_expr}\ (\lambda x : \texttt{Nat}.f\ x))\ (\texttt{to\_expr}\ (\lambda x : \texttt{Nat}.g\ x)).} \\
\texttt{Solution} & : & \textbf{Type}. \\
\texttt{All} & : & \texttt{Solution}. \\
\texttt{One} & : & \texttt{Nat} \longrightarrow \texttt{Nat} \longrightarrow \texttt{Solution}. \\
\texttt{None} & : & \texttt{Solution}. \\
\texttt{solve}\ (\texttt{mk\_expr}\ 0\ 0) & \hookrightarrow & \texttt{All}. \\
\texttt{solve}\ (\texttt{mk\_expr}\ 0\ (S\ n)) & \hookrightarrow & \texttt{None}. \\
\texttt{solve}\ (\texttt{mk\_expr}\ (S\ n)\ m) & \hookrightarrow & \texttt{One}\ m\ n.
\end{array}
$$

Figure 6: Parsing and solving linear equations

8 Conclusion

We have defined a notion of rewriting modulo β for the $\lambda\Pi$-calculus Modulo. We achieved this by encoding the $\lambda\Pi$-calculus Modulo into the framework of Higher-Order Rewrite Systems. As a consequence we also made available for the $\lambda\Pi$-calculus Modulo the confluence criteria designed for the HRSs (see for instance [14] or [15]). We proved that rewriting modulo β preserves typing. We generalized the criterion for product compatibility, by replacing the assumption of confluence by the confluence of the rewriting relation modulo β. We also generalized the criterion for well-typedness of rewrite rules to allow left-hand to be $\lambda\Pi$-patterns. These generalizations permit proving subject reduction and uniqueness of types for more systems.

A natural extension of this work would be to consider rewriting modulo $\beta\eta$ as in Higher-Order Rewrite Systems. This requires extending the conversion with η-reduction. But, as remarked in [10] (attributed to Nederpelt), $\rightarrow_{\beta\eta}$ is not confluent on untyped terms as the following example shows:

$$\lambda y : B.y \leftarrow_{\eta} \lambda x : A.(\lambda y : B.y)x \rightarrow_{\beta} \lambda x : A.x$$

Therefore properties such as product compatibility need to be proved another way. We leave this line of research for future work.

For the $\lambda\Pi$-calculus a notion of higher-order pattern matching has been proposed [16] based on Contextual Type Theory (CTT) [13]. This notion is similar to our. However, it is defined using the notion of meta-variable (which is native in CTT) instead of a translation into HRSs.

In [3], Blanqui studies the termination of the combination of β-reduction with a set of rewrite rules with matching modulo $\beta\eta$ in the polymorphic λ-calculus. His definition of rewriting modulo $\beta\eta$ is

direct and does not use any encoding. This leads to a slightly different notion a rewriting modulo β. For instance, $\mathtt{D}(\lambda : R.\mathtt{Exp}\ x)$ would reduce to $\mathtt{fMult}\ (\mathtt{D}\ (\lambda x : R.(\lambda y : R.y)\ x))\ (\lambda x : R.\mathtt{Exp}\ ((\lambda y : R.y)\ x))$ instead of $\mathtt{fMult}\ (\mathtt{D}\ (\lambda x : R.x))\ (\lambda x : R.\mathtt{Exp}\ x)$. It would be interesting to know whether the two definitions are equivalent with respect to confluence.

We implemented rewriting modulo β in Dedukti [5], our type-checker for the $\lambda\Pi$-calculus Modulo.

Acknowledgments. The author thanks very much Ali Assaf, Olivier Hermant, Pierre Jouvelot and the reviewers for their very careful reading and many suggestions.

References

[1] A. Assaf (2015): *A calculus of constructions with explicit subtyping*. In: *The 20th International Conference on Types for Proofs and Programs (TYPES '14)*.

[2] A. Assaf & G. Burel (2014): *Translating HOL to Dedukti*. Available at `https://hal.archives-ouvertes.fr/hal-01097412`.

[3] F. Blanqui (2015): *Termination of rewrite relations on lambda-terms based on Girard's notion of reducibility*. *Theoretical Computer Science*. To appear.

[4] M. Boespflug & G. Burel (2012): *CoqInE : Translating the calculus of inductive constructions into the $\lambda\Pi$-calculus modulo*. In: *The Second International Workshop on Proof Exchange for Theorem Proving (PxTP)*.

[5] M. Boespflug, Q. Carbonneaux, O. Hermant & R. Saillard: *Dedukti*. Available at `http://dedukti.gforge.inria.fr`.

[6] G. Burel (2013): *A Shallow Embedding of Resolution and Superposition Proofs into the $\lambda\Pi$-Calculus Modulo*. In: *The Third International Workshop on Proof Exchange for Theorem Proving (PxTP '13)*.

[7] R. Cauderlier & C. Dubois (2015): *Objects and Subtyping in the $\lambda\Pi$-Calculus Modulo*.

[8] D. Cousineau & G. Dowek (2007): *Embedding Pure Type Systems in $\lambda\Pi$-Calculus Modulo*. In: *The 8th International Conference on Typed Lambda Calculi and Applications (TLCA '07)*, doi:10.1007/978-3-540-73228-0_9.

[9] A. Dorra: *Equivalence de Curry-Howard entre le lambda-Pi calcul et la logique intuitionniste*. Report.

[10] H. Geuvers (1992): *The Church-Rosser Property for beta-eta-reduction in Typed lambda-Calculi*. In: *The Seventh Annual Symposium on Logic in Computer Science (LICS '92)*, doi:10.1109/LICS.1992.185556.

[11] D. Miller (1991): *A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification*. *Journal of Logic and Computation*, doi:10.1093/logcom/1.4.497.

[12] F. Müller (1992): *Confluence of the Lambda Calculus with Left-Linear Algebraic Rewriting*. *Information Processing Letters*, doi:10.1016/0020-0190(92)90155-O.

[13] Aleksandar Nanevski, Frank Pfenning & Brigitte Pientka (2008): *Contextual modal type theory*. *ACM Trans. Comput. Log.* 9(3), doi:10.1145/1352582.1352591.

[14] T. Nipkow (1991): *Higher-Order Critical Pairs*. In: *The Sixth Annual Symposium on Logic in Computer Science (LICS '91)*, doi:10.1109/LICS.1991.151658.

[15] V. van Oostrom (1995): *Development Closed Critical Pairs*. In: *The Second International Workshop on Higher-Order Algebra, Logic, and Term Rewriting, (HOA '95)*, doi:10.1007/3-540-61254-8_26.

[16] Brigitte Pientka (2008): *A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions*. In: *Symposium on Principles of Programming Languages, (POPL '08)*, doi:10.1145/1328438.1328483.

[17] R. Saillard (2013): *Towards explicit rewrite rules in the $\lambda\Pi$-calculus modulo*. In: *The 10th International Workshop on the Implementation of Logics (IWIL '13)*.