Towards explicit rewrite rules in the λΠ-calculus modulo

Ronan Saillard
MINES ParisTech
Fontainebleau, France
ronan.saillard@mines-paristech.fr

Abstract

This paper provides a new presentation of the λΠ-calculus modulo where the addition of rewrite rules is made explicit. The λΠ-calculus modulo is a variant of the λ-calculus with dependent types where β-reduction is extended with user-defined rewrite rules. Its expressiveness makes it suitable to serve as an output language for theorem provers, certified development tools or proof assistants. Addition of rewrite rules becomes an iterative process and rules previously added can be used to type new rules. We also discuss the condition rewrite rules must satisfy in order to preserve the Subject Reduction property and we give a criterion weaker than the usual one. Finally we describe the new version of Dedukti, a type-checker for the λΠ-calculus modulo for which we assess its efficiency in comparison with Coq, Twelf and Maude.

1 Introduction

Since the pioneering works on Automath [6], the λΠ-calculus (a.k.a. LF or λP) has been used to specify logics and programming languages [13]. In [10], Dowek and Cousineau suggested that extending this language with rewrite rules would greatly increase its expressiveness. The resulting language, called λΠ-calculus modulo, has since been used as an output language for theorem provers [7] [11] as well as for certified development tools [8] or proof assistants [2].

In this paper we provide a new presentation of the λΠ-calculus modulo that makes explicit the addition in the system of rewrite rules and we discuss the typing conditions they must satisfy. We also describe our new version of Dedukti, a type-checker for the λΠ-calculus modulo, and compare its performance with Coq [13], Twelf [13] and Maude [9].

2 The λΠ-calculus modulo

2.1 Language Description

In this section we describe the λΠ-calculus modulo, a variant of the λ-calculus with dependent types (λΠ-calculus) where β-reduction is extended with user-defined rewrite rules. The syntax of the λΠ-calculus modulo for terms, rewrite rules and contexts is given in Figure 1. A term is either a variable, an application, a λ-abstraction, a product type or a sort (either Type or Kind). A typing context is a list of declarations, i.e., pairs of a variable name and a term (its type). A rewrite rule is a triple made of a typing context and two terms. Finally a context is a list of declarations and rewrite rules.

As contexts may contain rewrite rules, they define a rewriting system. Given a context Γ we write →Γ the smallest relation, compatible with the structure of terms, such that for any rule [Δ] l → r in Γ and substitution σ with dom(σ) = dom(Δ), σl →Γ σr. We note →βΓ for →β ∪ →Γ and ≡βΓ for the congruence generated by →βΓ.
Towards explicit rewrite rules in the \(\lambda\Pi\)-calculus modulo Ronan Saillard

\[ x \in V \text{ (an infinite set)} \quad \text{(Variable)} \]

\[ s ::= \text{Type} | \text{Kind} \quad \text{(Sort)} \]

\[ t, u, A, B ::= x | t\ u | \lambda x^A.t | \Pi x^A.B \mid s \quad \text{(Term)} \]

\[ \Delta ::= \emptyset \mid \Delta(x : t) \quad \text{(Typing Context)} \]

\[ \mathcal{R} ::= [\Delta] t \hookrightarrow u \quad \text{(Rewrite Rule)} \]

\[ \Gamma ::= \emptyset \mid \Gamma(x : t) \mid \Gamma \mathcal{R} \quad \text{(Context)} \]

Figure 1: Syntax of the \(\lambda\Pi\)-calculus modulo.

The inference rules for the \(\lambda\Pi\)-calculus modulo, defining well-formed contexts and well-typed terms, are given in Figure 2. One can notice that these rules differ from the rules of \(\lambda\Pi\)-calculus only in two points. First, the congruence in the \texttt{Conv} rule is extended from \(\equiv\) to \(\equiv_{\beta}\), allowing for more terms to be typed. Secondly, there is a new rule \texttt{Rw} which makes possible the addition of rewrite rules in the context, thus extending explicitly the considered rewrite system. This latter rule is a novelty with respect to previous formalizations. Addition of rewrite rules becomes an iterative process: rules previously added can be used to type new rules. We obtain this way a formalism that is close to its actual implementation in \texttt{Dedukti} where a rewrite rule can appear anywhere in the specification.

![Figure 2: Typing rules for \(\lambda\Pi\)-calculus modulo.](image)

2.2 Typing Rewrite Rules

When adding a rewrite rule in the context one must check that it is compatible with typing. More precisely we want the so-called Subject Reduction property to hold: the type of a term should be invariant by rewriting.

\textbf{Subject Reduction:} if \(\Gamma \vdash t : T\) and \(t \rightarrow_{\beta} t'\) then \(\Gamma \vdash t' : T\).

Unexpectedly, Subject Reduction for \(\rightarrow_{\beta}\) might fail in the presence of rewrite rules since one has to prove that \(\Pi x^A.B_1 \equiv_{\beta T} \Pi x^A.B_2\) implies \(A_1 \equiv_{\beta T} A_2\) and \(B_1 \equiv_{\beta T} B_2\). However this is provable if \(\rightarrow_{\beta T}\) is confluent since, in this case, both \(\Pi\)-types reduce to a common term \(\Pi x^A.B\) which witnesses that \(A_1 \equiv_{\beta T} A \equiv_{\beta T} A_2\) and \(B_1 \equiv_{\beta T} B \equiv_{\beta T} B_2\). Subject reduction for \(\rightarrow_{\tau}\) is usually ensured by requiring rewrite rules \(\mid \Delta \mid l \hookrightarrow r\) to enjoy the following property.
Towards explicit rewrite rules in the λII-calculus modulo
Ronan Saillard

3 Implementation

3.1 General Description

Our current version of the type-checker for the λII-calculus modulo is a complete rewrite of the previous version of Dedukti [5]. It is now entirely written in OCaml and consists in about a thousand lines of code. It implements the standard algorithm for type checking semi-full Pure Type Systems [15] as well as a term reduction machine inspired from Matita’s [1] adapted to rewrite rules.

\[\text{Conditions} 1:\text{ There exists } T \text{ such that } \Gamma \Delta \vdash l : T \text{ and } \Gamma \Delta \vdash r : T.\]

This condition is often found in the literature, including our previous work on Dedukti [5], but is actually too strong. Figure 3 shows an example where a rewrite rule does not verify the condition above while preserving subject reduction. Indeed, on the last line, the term \text{Last} \ n \ (\text{Cons} \ n2 \ a \ l) is well-typed if and only if \( n1 \equiv_{\beta} S \ n2 \), which is trivially false because \( n1 \) and \( n2 \) are variables. However we know that, if for some terms \( t1 \) and \( t2 \), the left-hand side of the rule \text{Last} \ t1 \ (\text{Cons} \ t2 \ a \ l) is well-typed then we have \( t1 \equiv_{\beta} S \ t2 \) and the right-hand side has the same type. Thus the Subject Reduction property is preserved. Of course, we would have introduced non-linearity, resulting in less efficient rewriting. Indeed, now a conversion test is needed to check if a term is rewritable. Moreover non-linear rewrite systems are more complex to study when it comes to proving confluence and strong normalization.

In [4] Blanqui remarked that for a rewrite rule \( [\Delta] l \leftrightarrow r \) a weaker condition is actually sufficient (and necessary) to ensure the subject reduction property.

\[\text{Condition 2: For any substitution } \sigma \in S \text{ (i.e. } \text{dom}(\sigma) = \Delta \text{ and } (x : A) \in \Delta \text{ implies } \Gamma \vdash \sigma x : \sigma A), \text{ if } \Gamma \vdash a l : A \text{ then } \Gamma \vdash \sigma r : A.\]

Our previous example verifies this condition. Thus this solution allows more rules to be added in a context and avoids unnecessary non linear rules. The downside of this approach is that the condition becomes undecidable. However it is easy to prove in many cases. For instance one might find by unification a substitution \( \rho \) such that if \( \sigma \in S \) then \( \sigma = \sigma_0 \circ \rho \) and test if \( \Gamma \Delta \vdash pl : T \) implies \( \Gamma \Delta' \vdash pr : T \) with \( \Delta' := \Delta \setminus (\text{dom}(\rho)).\)

\[\begin{align*}
(Nat : Type) \\
(Z : Nat) \\
(S : \Pi x Nat. Nat) \\
(A : Type) \\
(Listn : \Pi x Nat. Type) \\
(Nil : ListnZ) \\
(Cons : \Pi a. \Pi n Nat. \Pi x (Listn x). Listn (S n)) \\
(Last : \Pi a Nat. \Pi x (Listn x). A) \\
((n1 : Nat)(n2 : Nat)(a : A) \\
Last n1 (Cons n2 a Nil) \rightarrow a \\
((n1 : Nat)(n2 : Nat)(a : A)(l : List n2) \\
Last n1 (Cons n2 a l) \rightarrow Last n2 l)
\end{align*}\]

Figure 3: The function \text{Last} in the λII-calculus modulo and the corresponding source in Dedukti.
3.2 Benchmarks

We assessed the efficiency of Dedukti for both type-checking and rewriting. We compared the results with those of Coq (version 8.3pl4), a proof assistant for the calculus of inductive constructions, Twelf (version 1.7.1+), an implementation of the \( \lambda \Pi \)-calculus, and Maude (version 2.6), a high-level language which supports rewriting logic computation. The tests were run on a Linux laptop with a processor Intel Core i7-3520M CPU @ 2.90GHz x 4 and 16GB of Ram. The different benchmark files can be obtained on Dedukti’s website.

3.2.1 Dedukti vs. Coq vs. Twelf

We chose to use the OpenTheory \([12]\) library as a benchmark to assess type-checking performance. This library was conceived as a way to share theorems between theorem provers implementing higher-order logic. Its core is composed of more than 80 files of variable size (up to 250Mo) containing general definitions and theorems. Thanks to Holide \([2]\), the OpenTheory format can be encoded into the \( \lambda \Pi \)-calculus (i.e., without rewrite rule) and printed in Coq, Twelf or Dedukti’s input format. It can also be translated, via an encoding similar to \([10]\) and using one rewrite rule, in the \( \lambda \Pi \)-calculus modulo and printed in Dedukti’s input format. The latter translation is not available for Coq or Twelf as these tools cannot deal with user-defined rewrite rules. Some of the results are given Table 4. We can see that Dedukti is globally 20% faster than Coq and 30% faster than Twelf on the \( \lambda \Pi \)-calculus encoding. On the other hand type checking the \( \lambda \Pi \)-calculus modulo translation is more than 20 times faster. The explanation is that rewrite rules permit to obtain smaller terms that moreover have less type dependencies thus requiring less conversion tests while type checking.

```
<table>
<thead>
<tr>
<th>File</th>
<th>Size</th>
<th>Dedukti (( \lambda \Pi ))</th>
<th>Coq (( \lambda \Pi ))</th>
<th>Twelf (( \lambda \Pi ))</th>
<th>Dedukti (( \lambda \Pi )m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>natural-exp-thm.dk</td>
<td>55M</td>
<td>11</td>
<td>13</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>list-def.dk</td>
<td>84M</td>
<td>21</td>
<td>20</td>
<td>32</td>
<td>1</td>
</tr>
<tr>
<td>set-thm.dk</td>
<td>97M</td>
<td>30</td>
<td>63</td>
<td>33</td>
<td>2</td>
</tr>
<tr>
<td>relation-natural-thm.dk</td>
<td>122M</td>
<td>53</td>
<td>40</td>
<td>56</td>
<td>2</td>
</tr>
<tr>
<td>real-def.dk</td>
<td>259M</td>
<td>53</td>
<td>107 (1mn47)</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td>All files (88)</td>
<td>1.4G</td>
<td>396 (6mn35)</td>
<td>470 (7mn50)</td>
<td>523 (8mn43)</td>
<td>17</td>
</tr>
</tbody>
</table>
```

Figure 4: Type-checking time (in seconds).

3.2.2 Dedukti vs Maude

As a preliminary test of Dedukti’s rewriting capabilities, we ran both Dedukti and Maude on a series of handwritten benchmarks consisting on computations involving arithmetic functions defined as algebraic rewrite rules on unary integers. Results are given in Table 5. These tests show no clear correspondence between time spent by Dedukti and Maude. Further work should be done to investigate the differences between the rewrite strategy used by both tools.

4 Conclusion

We have presented a version of the \( \lambda \Pi \)-calculus modulo where rewrite rules are explicitly added in contexts making their addition an iterative process where previous rules can be used to type
Towards explicit rewrite rules in the $\lambda\Pi$-calculus modulo

Ronan Saillard

<table>
<thead>
<tr>
<th>Expression</th>
<th>Dedukti</th>
<th>Maude</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{10}$</td>
<td>31</td>
<td>6</td>
</tr>
<tr>
<td>$3^6$</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>$5 \times 4^5$</td>
<td>56</td>
<td>53</td>
</tr>
<tr>
<td>$(10 \times 10) \times (10 \times 10)$</td>
<td>46</td>
<td>28</td>
</tr>
</tbody>
</table>

Figure 5: Rewriting time (in seconds).

the new ones. We have discussed the condition rewrite rules must satisfy in order to preserve the Subject Reduction property and we have given a criterion weaker than the usual one. We have described the new version of Dedukti, and we have shown that it is faster than Coq and Twelf. We have also given preliminary comparison data with Maude.

At present our type-checking algorithm assumes the confluence (for completeness) and the strong normalization (for termination) of $\rightarrow_{\beta\eta}$, As only algebraic linear rewrite rules are implemented in Dedukti, the confluence of $\rightarrow_{\beta\eta}$ follows from the confluence of $\rightarrow_\Gamma$ [16]. Regarding the strong normalization of the combination of $\beta$-reduction with an arbitrary rewrite system, general criteria exist in the literature. See for instance [3] for object-level rewriting or [4] for object-level and type-level rewriting. We plan to integrate these works into Dedukti in a future release.

References