universität innsbruck

Seminar Report

ProTeM

Christina Kohl
christina.kohl@student.uibk.ac.at

23 February 2018

Supervisor: Prof. Dr. Aart Middeldorp

Abstract

Proof terms are a useful concept for reasoning about computations in term rewriting. Human calculation with proof terms is tedious and error-prone. We present ProTeM, a new tool that offers support for manipulating proof terms that represent multisteps in left-linear rewrite systems.

Contents

1 Introduction

Proof terms represent computations in term rewriting. They were introduced by van Oostrom and de Vrijer for first-order left-linear rewrite systems to study equivalence of reductions in [17] and [13, Chapter 8]. Extensions to higher-order rewriting and infinitary rewriting are reported in [1] and [6], respectively. Hirokawa and Middeldorp used proof terms for confluence analysis of left-linear rewrite systems [2, 3].

Our motivation for studying proof terms is to close an important gap between proofs produced by automatic confluence checkers and *certified* proofs. Numerous confluence criteria described in the literature have been formalized in IsaFor, a large Isabelle/HOL library for term rewriting, see [9] for a recent overview. This includes the well-known result of Huet [4] stating that a left-linear rewrite system is confluent if its critical pairs are closed by a parallel step [10]. Its extension to multisteps (also called development steps) by van Oostrom [16] thus far escaped all attempts to obtain a formalized proof. The picture proof in [16] conveys the intuition but is very hard to formalize in a modern proof assistant. We believe that proof terms together with residual theory [13, Section 8.7] will help to close the gap.

Calculations with proof terms are tedious and error-prone to do by hand, which is why we developed ProTeM. Besides providing basic operations for manipulating proof terms that represent multisteps in left-linear rewrite systems, like join and residual, ProTeM supports new operations on proof terms that are required for a formalized proof of the main result of [16]. The latter include an inductive definition for computing the amount of overlap and a function that returns the critical overlaps between co-initial proof terms.

In the next section we recall proof terms and introduce new operations for measuring overlap between two proof terms. The web interface of ProTeM is described in Section 3 and in Section 4 we present some implementation details. We conclude in Section 5 with ideas for future extensions of ProTeM.

2 Proof Terms

Proof terms are built from function symbols, variables, and rule symbols. The latter represent rewrite rules and have a fixed arity which is the number of different variables in the represented rule. We use Greek letters as rule symbols.

Example 2.1. Consider the following TRS consisting of five rewrite rules which are associated with rule symbols α to ε:

$$\begin{aligned}
\alpha&: && \mathsf{f}(\mathsf{g}(x)) \to \mathsf{g}(\mathsf{h}(x, \mathsf{i}(\mathsf{a})))\\
\beta&: && \mathsf{g}(\mathsf{h}(\mathsf{h}(\mathsf{i}(x), y), \mathsf{f}(z))) \to \mathsf{h}(\mathsf{h}(y, y), \mathsf{f}(z))\\
\gamma&: && \mathsf{i}(x) \to x\\
\delta&: && \mathsf{h}(x, \mathsf{f}(y)) \to \mathsf{h}(\mathsf{i}(\mathsf{f}(y)), \mathsf{f}(y))\\
\varepsilon&: && \mathsf{g}(\mathsf{h}(x, y)) \to \mathsf{h}(x, y)
\end{aligned}$$

Two possible proof terms in this system are A and B:

$$A = \alpha(\mathsf{h}(\delta(\mathsf{i}(\gamma(\mathsf{a})), \mathsf{i}(\mathsf{a})), \alpha(\mathsf{a})))$$
$$B = \mathsf{f}(\beta(\mathsf{i}(\mathsf{a}), \mathsf{f}(\gamma(\mathsf{a})), \mathsf{g}(\mathsf{a})))$$

In this section we present the operations on proof terms that are implemented in ProTeM. We will start by giving a few basic definitions that apply to rule symbols of a given rewrite system.

Definition 2.2. If α is a rule symbol then $\mathsf{lhs}(\alpha)$ ($\mathsf{rhs}(\alpha)$) denotes the left-hand (right-hand) side of the rewrite rule represented by α. Furthermore $\mathsf{var}(\alpha)$ denotes the list $(x_1, \ldots, x_n)$ of variables appearing in α in some fixed order. The length of this list is the arity of α.

For the following definitions we need a notation for substituting terms for the variables occuring in a rewrite rule.

Definition 2.3. Given a rule symbol α with $\mathsf{var}(\alpha) = (x_1, \ldots, x_n)$ and terms $t_1, \ldots, t_n$, we write $\langle t_1, \ldots, t_n \rangle_\alpha$ for the substitution $\{x_i \mapsto t_i \mid 1 \leqslant i \leqslant n\}$.

We mentioned in the introduction that proof terms represent multisteps in a (left-linear) rewrite system. To be precise a proof term A witnesses a multistep from its source to its target.

Definition 2.4. Given a proof term A, its source $\mathsf{src}(A)$ and target $\mathsf{tgt}(A)$ are computed by the following clauses:

$$\begin{aligned}
\mathsf{src}(x) &= \mathsf{tgt}(x) = x \\
\mathsf{src}(f(A_1, \ldots, A_n)) &= f(\mathsf{src}(A_1), \ldots, \mathsf{src}(A_n)) \\
\mathsf{src}(\alpha(A_1, \ldots, A_n)) &= \mathsf{lhs}(\alpha)\langle \mathsf{src}(A_1), \ldots, \mathsf{src}(A_n) \rangle_\alpha \\
\mathsf{tgt}(f(A_1, \ldots, A_n)) &= f(\mathsf{tgt}(A_1), \ldots, \mathsf{tgt}(A_n)) \\
\mathsf{tgt}(\alpha(A_1, \ldots, A_n)) &= \mathsf{rhs}(\alpha)\langle \mathsf{tgt}(A_1), \ldots, \mathsf{tgt}(A_n) \rangle_\alpha
\end{aligned}$$

Proof terms A and B are *co-initial* if they have the same source.

Example 2.5. Consider again the two proof terms A and B from Example 2.1. By computing their respective sources we can see that they are indeed co-initial:

$$\mathsf{src}(A) = \mathsf{src}(B) = \mathsf{f}(\mathsf{g}(\mathsf{h}(\mathsf{h}(\mathsf{i}(\mathsf{i}(\mathsf{a})), \mathsf{f}(\mathsf{i}(\mathsf{a}))), \mathsf{f}(\mathsf{g}(\mathsf{a})))))$$

Their targets are

$$\begin{aligned}
\mathsf{tgt}(A) &= \mathsf{g}(\mathsf{h}(\mathsf{h}(\mathsf{h}(\mathsf{i}(\mathsf{a}), \mathsf{i}(\mathsf{a})), \mathsf{g}(\mathsf{h}(\mathsf{a}, \mathsf{i}(\mathsf{a})))), \mathsf{i}(\mathsf{a}))) \\
\mathsf{tgt}(B) &= \mathsf{f}(\mathsf{h}(\mathsf{h}(\mathsf{f}(\mathsf{a}), \mathsf{f}(\mathsf{a})), \mathsf{f}(\mathsf{g}(\mathsf{a})))).
\end{aligned}$$

To further illustrate the connection between proof terms and multisteps, Figure 1 shows a tree representation of $s = \mathsf{src}(A) = \mathsf{src}(B)$, where the redexes in A (B) have been marked in red (green). It is easy to see that A and B are overlapping at three positions. We will verify this in the next section when we formally define a function to determine the amount of overlap between two proof terms.

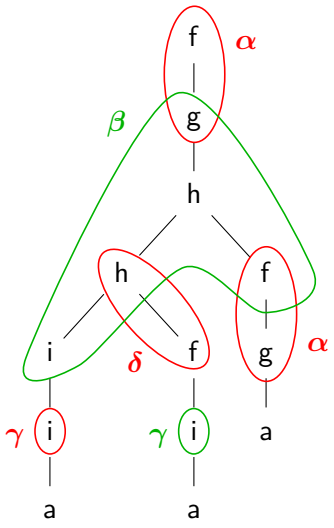

Figure 1: The term $\mathsf{f}(\mathsf{g}(\mathsf{h}(\mathsf{h}(\mathsf{i}(\mathsf{i}(\mathsf{a})),\mathsf{f}(\mathsf{i}(\mathsf{a}))),\mathsf{f}(\mathsf{g}(\mathsf{a})))))$ with overlapping redexes.

2.1 Basic Operations

Below we give definitions of four of the most basic operations on proof terms: determining orthogonality, joining two proof terms, computing the residual of two proof terms, and deleting steps of a proof term. When two proof terms have no overlap in their redexes we call them orthogonal.

Definition 2.6. The *orthogonality* predicate $A \perp B$ on two proof terms A and B is defined by the following clauses:

$$\begin{aligned} x \perp x & \\ f(A_1,\ldots,A_n) \perp f(B_1,\ldots,B_n) &\iff A_i \perp B_i \text{ for all } 1 \leqslant i \leqslant n \\ \alpha(A_1,\ldots,A_n) \perp \mathsf{lhs}(\alpha)\langle B_1,\ldots,B_n\rangle_\alpha &\iff A_i \perp B_i \text{ for all } 1 \leqslant i \leqslant n \\ \mathsf{lhs}(\alpha)\langle A_1,\ldots,A_n\rangle_\alpha \perp \alpha(B_1,\ldots,B_n) &\iff A_i \perp B_i \text{ for all } 1 \leqslant i \leqslant n \end{aligned}$$

In all other cases $A \perp B$ is false.

Example 2.7. Consider the TRS consisting of the three rewrite rules

$$\alpha\colon \mathsf{f}(\mathsf{a},x,y) \to \mathsf{g}(x,x,y) \qquad \beta\colon \mathsf{a} \to \mathsf{b} \qquad \gamma\colon \mathsf{h}(x) \to \mathsf{h}(\mathsf{h}(x))$$

together with the three co-initial proof terms

$$A = \mathsf{f}(\beta,\mathsf{a},\mathsf{h}(\mathsf{a})) \qquad B = \mathsf{f}(\mathsf{a},\beta,\mathsf{h}(\mathsf{a})) \qquad C = \alpha(\mathsf{a},\gamma(\mathsf{a})).$$

Here we have $A \perp B$ and $B \perp C$ but $A \not\perp C$.

The next operation, join, combines two co-initial proof terms into one single proof term.

Definition 2.8. The *join* operation $A \sqcup B$ on two co-initial proof terms A and B is defined by the following clauses:

$$\begin{aligned} x \sqcup x &= x \\ f(A_1, \ldots, A_n) \sqcup f(B_1, \ldots, B_n) &= f(A_1 \sqcup B_1, \ldots, A_n \sqcup B_n) \\ \alpha(A_1, \ldots, A_n) \sqcup \alpha(B_1, \ldots, B_n) &= \alpha(A_1 \sqcup B_1, \ldots, A_n \sqcup B_n) \\ \alpha(A_1, \ldots, A_n) \sqcup \mathsf{lhs}(\alpha)\langle B_1, \ldots, B_n\rangle_\alpha &= \alpha(A_1 \sqcup B_1, \ldots, A_n \sqcup B_n) \\ \mathsf{lhs}(\alpha)\langle A_1, \ldots, A_n\rangle_\alpha \sqcup \alpha(B_1, \ldots, B_n) &= \alpha(A_1 \sqcup B_1, \ldots, A_n \sqcup B_n) \end{aligned}$$

Example 2.9. Consider again the proof terms A, B, and C of Example 2.7. We get $A \sqcup B = \mathsf{f}(\beta, \beta, \mathsf{h}(\mathsf{a}))$ and $B \sqcup C = \alpha(\beta, \gamma(\mathsf{a}))$.

Next we define the residual of a proof term A after applying B.

Definition 2.10. The *residual* operation $A \,/\, B$ on two co-initial proof terms A and B is defined by the following clauses:

$$\begin{aligned} x \,/\, x &= x \\ f(A_1, \ldots, A_n) \,/\, f(B_1, \ldots, B_n) &= f(A_1 \,/\, B_1, \ldots, A_n \,/\, B_n) \\ \alpha(A_1, \ldots, A_n) \,/\, \alpha(B_1, \ldots, B_n) &= \mathsf{rhs}(\alpha)\langle A_1 \,/\, B_1, \ldots, A_n \,/\, B_n\rangle_\alpha \\ \alpha(A_1, \ldots, A_n) \,/\, \mathsf{lhs}(\alpha)\langle B_1, \ldots, B_n\rangle_\alpha &= \alpha(A_1 \,/\, B_1, \ldots, A_n \,/\, B_n) \\ \mathsf{lhs}(\alpha)\langle A_1, \ldots, A_n\rangle_\alpha \,/\, \alpha(B_1, \ldots, B_n) &= \mathsf{rhs}(\alpha)\langle A_1 \,/\, B_1, \ldots, A_n \,/\, B_n\rangle_\alpha \end{aligned}$$

Join and residual are partial operations. They are well-defined when $A \perp B$ holds.

Example 2.11. Consider again the proof terms A and B, and C of Example 2.7. Since $A \perp B$ and $B \perp C$ we can compute the residuals $A \,/\, B$ and $B \,/\, A$, as well as $B \,/\, C$ and $C \,/\, B$:

$$\begin{aligned} A \,/\, B &= \mathsf{f}(\beta, \mathsf{b}, \mathsf{h}(\mathsf{a})) & B \,/\, A &= \mathsf{f}(\mathsf{b}, \beta, \mathsf{h}(\mathsf{a})) \\ B \,/\, C &= \mathsf{g}(\mathsf{b}, \mathsf{b}, \mathsf{h}(\mathsf{h}(\mathsf{a}))) & C \,/\, B &= \alpha(\mathsf{b}, \gamma(\mathsf{a})) \end{aligned}$$

Finally we define the deletion $A - B$, which is used to remove the steps in B from A.

Definition 2.12. The *deletion* operation $A - B$ on two co-initial proof terms A and B is defined by the following clauses:

$$\begin{aligned} x - x &= x \\ f(A_1, \ldots, A_n) - f(B_1, \ldots, B_n) &= f(A_1 - B_1, \ldots, A_n - B_n) \\ \alpha(A_1, \ldots, A_n) - \alpha(B_1, \ldots, B_n) &= \mathsf{lhs}(\alpha)\langle A_1 - B_1, \ldots, A_n - B_n\rangle_\alpha \\ \alpha(A_1, \ldots, A_n) - \mathsf{lhs}(\alpha)\langle B_1, \ldots, B_n\rangle_\alpha &= \alpha(A_1 - B_1, \ldots, A_n - B_n) \end{aligned}$$

Example 2.13. For the three proof terms A, B, C of Example 2.7 only the deletion $C - B$ is defined:

$$C - B = \alpha(\mathsf{a}, \gamma(\mathsf{a}))$$

2.2 Measuring Overlap

An important concept in the correctness proof of the confluence theorem in [16] is the amount of overlap between two multisteps. Below we present an inductive definition for measuring the overlap between co-initial proof terms. It is based on a special labeling of the source of a proof term.

Definition 2.14. We write $\mathsf{lhs}^\sharp(\alpha)$ for the result of labeling every function symbol in $\mathsf{lhs}(\alpha)$ with α as well as the distance to the root of α:

$$\varphi(t,\alpha,i) = \begin{cases} t & \text{if } t \in \mathcal{V} \\ f_{\alpha^i}(\varphi(t_1,\alpha,i+1),\ldots,\varphi(t_n,\alpha,i+1)) & \text{if } t = f(t_1,\ldots,t_n) \end{cases}$$

Definition 2.15. The mapping $\mathsf{src}^\sharp$ computes the labeled source of a proof term:

$$\begin{aligned} \mathsf{src}^\sharp(x) &= x \\ \mathsf{src}^\sharp(f(A_1,\ldots,A_n)) &= f(\mathsf{src}^\sharp(A_1),\ldots,\mathsf{src}^\sharp(A_n)) \\ \mathsf{src}^\sharp(\alpha(A_1,\ldots,A_n)) &= \mathsf{lhs}^\sharp(\alpha)\langle \mathsf{src}^\sharp(A_1),\ldots,\mathsf{src}^\sharp(A_n)\rangle_\alpha \end{aligned}$$

Example 2.16. For the proof terms A and B of Example 2.1 we have:

$$\begin{aligned} \mathsf{src}^\sharp(A) &= \mathsf{f}_{\alpha^0}(\mathsf{g}_{\alpha^1}(\mathsf{h}(\mathsf{h}_{\delta^0}(\mathsf{i}(\mathsf{i}_{\gamma^0}(\mathsf{a})),\mathsf{f}_{\delta^1}(\mathsf{i}(\mathsf{a}))),\mathsf{f}_{\alpha^0}(\mathsf{g}_{\alpha^1}(\mathsf{a})))))\\ \mathsf{src}^\sharp(B) &= \mathsf{f}(\mathsf{g}_{\beta^0}(\mathsf{h}_{\beta^1}(\mathsf{h}_{\beta^2}(\mathsf{i}_{\beta^3}(\mathsf{i}(\mathsf{a})),\mathsf{f}(\mathsf{i}_{\gamma^0}(\mathsf{a}))),\mathsf{f}_{\beta^2}(\mathsf{g}(\mathsf{a}))))) \end{aligned}$$

Given two co-initial proof terms A and B, the following function computes a single labeled term in which all function symbols corresponding to redex patterns in A and B are marked.

Definition 2.17. For two co-initial proof terms A and B we define the *merge* operation:

$$\mathsf{merge}(A,B) = \mathsf{merge}'(\mathsf{src}^\sharp(A),\mathsf{src}^\sharp(B))$$

with $\mathsf{merge}'(s,t) = s$ for $s,t \in \mathcal{V}$ and $\mathsf{merge}'(s,t) = f_{ab}(\mathsf{merge}'(s_1,t_1),\ldots,\mathsf{merge}'(s_n,t_n))$ if $s = f_a(s_1,\ldots,s_n)$ and $t = f_b(t_1,\ldots,t_n)$. Here we identify an unlabeled function symbol f with $f_\_$.

Example 2.18. For the two co-initial proof terms A and B of Example 2.1 we have:

$$\mathsf{merge}(A,B) = \mathsf{f}_{\alpha^0\_}(\mathsf{g}_{\alpha^1\beta^0}(\mathsf{h}_{\_\beta^1}(\mathsf{h}_{\delta^0\beta^2}(\mathsf{i}_{\_\beta^3}(\mathsf{i}_{\gamma^0\_}(\mathsf{a})),\mathsf{f}_{\delta^1\_}(\mathsf{i}_{\_\gamma^0}(\mathsf{a}))),\mathsf{f}_{\alpha^0\beta^2}(\mathsf{g}_{\alpha^1\_}(\mathsf{a})))))$$

Here a abbreviates $\mathsf{a}_{\_\_}$.

Definition 2.19. The function ▲ uses merge to measure the amount of overlap between two co-initial proof terms A and B:

▲$(A,B) = \mathsf{measure}(\mathsf{merge}(A,B))$ with $\mathsf{measure}(u) = 0$ if $u \in \mathcal{V}$ and

$$\mathsf{measure}(f_{a^kb^l}(u_1,\ldots,u_n)) = \begin{cases} 1 + \sum_{i=1}^{n} \mathsf{measure}(u_i) & \text{if } a^k \neq - \text{ and } b^l \neq - \\ \sum_{i=1}^{n} \mathsf{measure}(u_i) & \text{otherwise} \end{cases}$$

Example 2.20. Continuing Example 2.18 we compute the amount of overlap between A and B of Example 2.1:

$$▲(A, B) = 3$$

Definition 2.21. The overlaps function collects all pairs of overlapping redexes in two co-initial proof terms A and B:

$$\mathsf{overlaps}(A,B) = \left\{ (p,\alpha,q,\beta) \;\middle|\; \begin{array}{l} p,q \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(u),\ \ell_1(u(p)) = \alpha^0,\ \ell_2(u(q)) = \beta^0, \text{ and either} \\ p \leqslant q \text{ and } \ell_1(u(q)) = \alpha^{|q\backslash p|} \text{ or } q < p \text{ and } \ell_2(u(p)) = \beta^{|p\backslash q|} \end{array} \right\}$$

Here $\ell_1(f_{ab}) = a$, $\ell_2(f_{ab}) = b$, and $u = \mathsf{merge}(A, B)$.

The condition $\ell_1(u(q)) = \alpha^{|q\backslash p|}$ in the first case of the definition of $\mathsf{overlaps}(A, B)$ ensures that $q\backslash p$ is a position in $\mathsf{lhs}(\alpha)$.

Example 2.22. we compute the list of overlaps between proof terms A and B of Example 2.1:

$$\mathsf{overlaps}(A, B) = \{(\epsilon, \boldsymbol{\alpha}, 1, \boldsymbol{\beta}), (111, \boldsymbol{\delta}, 1, \boldsymbol{\beta}), (112, \boldsymbol{\alpha}, 1, \boldsymbol{\beta})\}$$

As predicted by $▲(A, B) = 3$ the length of this list is three.

3 Web Interface

In this section we will first give a brief overview of the main parts of ProTeM's user interface, followed by a more detailed description of each of its features. The web interface of ProTeM can be accessed at

`http://informatik-protem.uibk.ac.at:8080/protem/`

The layout of our application is displayed in Figure 2. At the center of the screen we have a large area for displaying the history of commands a user has entered (on the left), together with result output corresponding to these commands (on the right). Below that there is a smaller panel where all rules of the currently loaded term rewrite system are displayed. At the bottom of the screen we have a command line with several buttons above it, that help users enter unusual symbols such as Greek letters for rule symbols or the $\perp$ symbol for the orthogonality predicate on proof terms. To the left of the screen we have a sidebar that gives an overview of the syntax that is used for commands.

3.1 Uploading a Term Rewrite System

When first opening the website, a simple example rewrite system is loaded per default. Users can upload their own rewrite systems from `.trs` files. The files need to correspond to a simplified form of the standard TRS-format as described in [7], where only the `VAR` and `RULES` sections are taken into account. Additionally the rule symbols ProTeM should use can be specified in the file by prepending each rule with its corresponding symbol

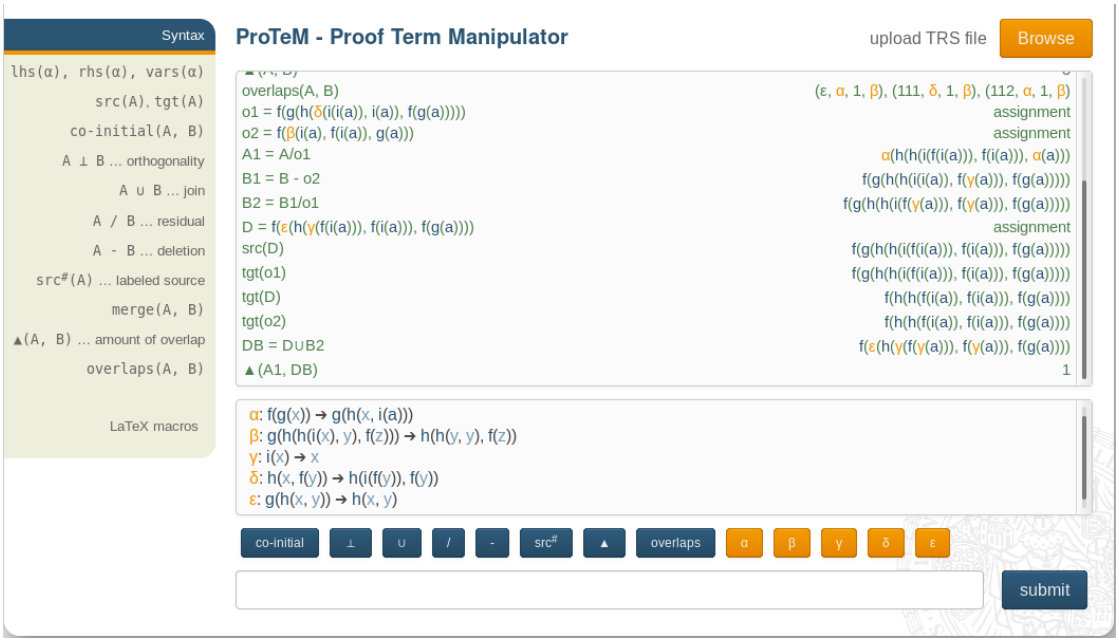

Figure 2: Screenshot of a ProTeM session.

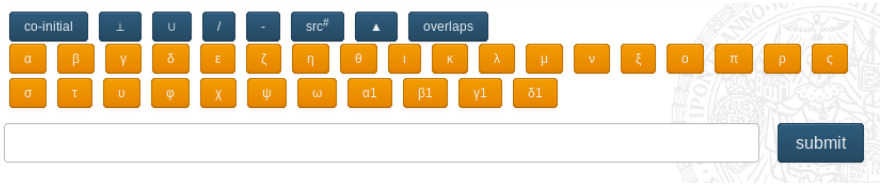

Figure 3: The adapted symbol buttons after uploading a TRS with 28 rules.

followed by a colon. If one or more rules have no specified rule symbols, ProTeM chooses a new Greek letter for each rule, starting from α. If a rewrite system contains more than 24 rules (24 is the number of letters in the Greek alphabet) ProTeM will assign $\alpha 1$ to the 25th rule, $\beta 1$ to the 26th rule, and so on. When uploading a new rewrite system, the buttons above the command line will automatically change according to the new rule symbols. Figure 3 shows the list of buttons after uploading a rewrite system with 28 rules.

3.2 Commands

There are two types of commands available, one are assignments, the other computations on proof terms. Assignments have syntax `id = proofterm` where `id` can be any string and `proofterm` any valid proof term. Notably it is possible to use results of computations in assignments (e.g. $C = B - \mathsf{f}(\beta(\mathsf{i}(\mathsf{a}), \mathsf{f}(\mathsf{i}(\mathsf{a})), \mathsf{g}(\mathsf{a})))$, see also Figure 2). Commands for rule symbols are `lhs`(α), `rhs`(α) and `vars`(α) where α can be any rule symbol used in

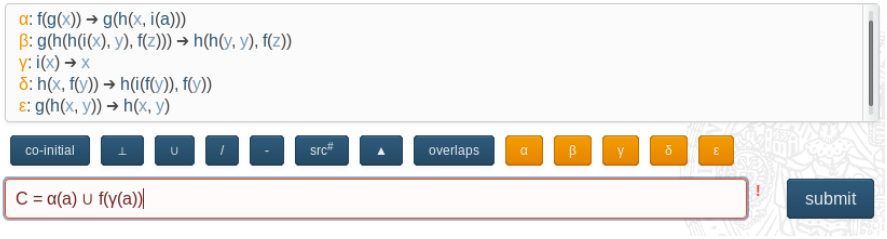

Figure 4: An invalid assignment; the join operation of these two proof terms is not defined.

the current TRS. Commands for proof terms include all operations described in Section 2. Their syntax is listed in the sidebar of our application.

Commands have to be entered into the text field at the bottom of the screen. The blue buttons above it can be used to enter special symbols that are used for some of the commands (like $\perp$ for orthogonality, or ▲ for measuring the amount of overlap between two proof terms). In addition there is one orange button for each currently used rule symbol. When pressing one of the buttons, the corresponding symbol appears in the command line, with the focus returning immediately to the text field itself so that the user can carry on typing. A command can be submitted either by pressing enter or by using the "Submit" button. If the command line contains a valid command, it will be sent to the server and executed. The result will then be displayed in the output area above. If the command was not valid (e.g. trying to assign the result of an undefined operation), an error will be displayed (Figure 4). In Section 4.3 we will explain how commands are processed internally.

3.3 Export to LaTeX

A proof term or labeled proof term can be exported as a LaTeX string. To correctly insert proof terms from ProTeM into a LaTeX document it is first necessary to add the required macros. These define colors and provide support for UTF8 encoding of Greek letters. In particular we define three new commands `\pfun`, `\pvar`, `\prule` which determine the representations of function symbols, variables and rule symbols respectively. The macros can be downloaded by clicking on the "LaTeX macros" entry in the sidebar. Clicking on any proof term in the output area will open a popup view, which contains a text field with the LaTeX representation of that proof term (Figure 5). It can then be copy-and-pasted into any document. The proof terms in this report have been created using this feature.

4 Implementation Details

The core functionality of ProTeM is written in Scala. For the web component we used the Vaadin framework [14]. Vaadin is a Java web application framework that makes it easier for developers who don't have much experience with web technologies, such as JavaScript, HTML and HTTP requests, to design responsive and interactive web

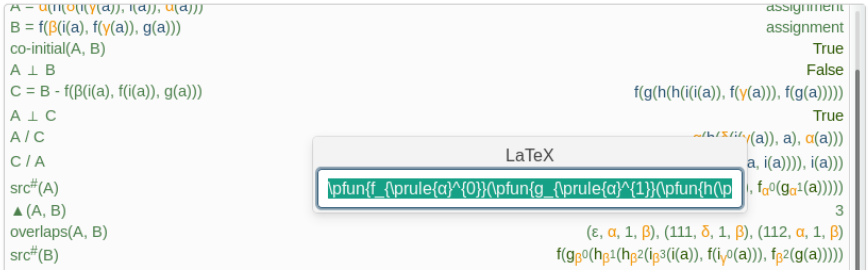

Figure 5: Popup view containing the LaTeX representation of a labeled proof term.

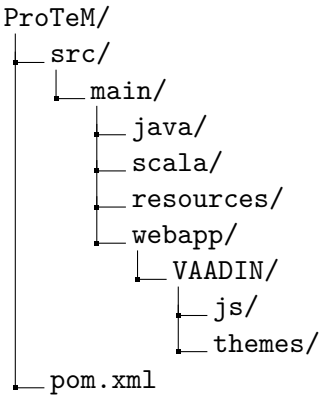

Figure 6: The project structure of ProTeM.

applications. Vaadin allows developers to write all required code in pure Java (or any other language that runs on the JVM). Applications can also be extended with custom HTML or JavaScript and themed with CSS. From a technological point of view the UI logic of a Vaadin application runs as a Java Servlet in a Java application server. ProTeM for example is hosted on a Jetty [5] server. On the client side Vaadin uses JavaScript to render the user interface in the browser and communicate user events to the server. All communication is automated and makes heavy use of AJAX (Asynchronous JavaScript and XML) to make applications as responsive as possible. An additional benefit for our particular application was that Vaadin automatically stores the state of each user session (as long as the browser window is open), so that we can provide users with an interactive interface and still call our Scala functions on the server for all computations on proof terms.

4.1 Project Structure

Our project is composed of Scala and Java source code as well as HTML layout definitions and CSS style definitions. The project structure is displayed in Figure 6. The Vaadin Framework core library and all Vaadin add-ons are available through Maven [8], a popular build and dependency management system. New Vaadin projects can be created from

Maven archetypes, which are basically project templates. For ProTeM we used the `vaadin-archetype-application` [15] as our starting point. In Maven the project and configuration details are defined in an XML file called `pom.xml` which is located in the root folder of the project. The default `pom.xml` that comes with the Vaadin archetype however does not support Scala code compilation. For that we had to add Scala dependencies and the scala-maven-plugin [12].

As can be seen in Figure 6 our main source folder is divided into two subfolders, one for Scala source code and one for Java source code. The `scala` folder contains the main functionality for proof term manipulations whereas the `java` folder contains only UI specific code. The `resources` folder contains the example trs file that is loaded per default into the app as well as the file `proofterm_macros.tex` which contains our LaTeX macros for exporting proof terms. An important component of every Vaadin application is the `VAADIN` folder inside `webapp` which contains theme definitions for the app. Themes can either be defined in pure CSS or Sass (Syntactically Awesome Stylesheets [11]). In addition the `VAADIN` folder can contain Java scripts and custom HTML layouts.

4.2 UI Layout

The main layout of our user interface is defined in the file `index.html` (located in `VAADIN/themes/cl/layouts`). It provides the CL typical header and a sidebar with our syntax descriptions. In addition it contains a special `div` component with a `data-location` property. This `div` can be accessed and filled with content from inside a Java class. On the Java side we only need to extend Vaadins abstract class `UI` and override its `init` method. There we can load `index.html` as a `CustomLayout`, create all our dynamic and interactive layout parts and add them to `index.html`. The code snippet in Listing 1 illustrates the necessary steps. Creating layout components in Java allows us to attach simple event listeners to them, which in turn can access all proof term methods located inside the `scala` folder. We will describe the interaction between the different components of our app in more detail in the next section, when we explain how a command entered into the command line is processed internally.

4.3 Processing Commands

To understand how ProTeM works internally we will look at the implementation of its core functionality, parsing and executing commands. An important concept to understand is how Vaadin treats session states internally. For each new session (e.g. a new ProTeM tab in a browser) Vaadin creates a new UI instance. In our case a new instance of `ProTeMUI` is created. We give each instance of `ProTeMUI` its own `State` object, which contains all the necessary data for a ProTeM session, including the currently loaded TRS and all assignments.

We already mentioned that each interactive component can have a listener attached to it in Java. In particular we have one listener for clicks on the “Submit” button beside the command line and another listener attached to the command line itself, which gets notified when enter is pressed. Both listeners simply call the method `processCommand()`

```
@Override
protected void init(VaadinRequest vaadinRequest) {
  getPage().setTitle(
  "ProTeM␣|␣Computational␣Logic␣|␣␣University␣of␣Innsbruck");
  // load index.html from layouts
  CustomLayout index = new CustomLayout("index");
  // the main layout component
  final VerticalLayout layout = new VerticalLayout();

  ... // initialize the individual layout components

  layout.addComponent(uploadLayout);
  layout.addComponent(outputArea);
  layout.addComponent(trsArea);
  layout.addComponent(buttonLayout);
  layout.addComponent(commandArea);

  // add the main layout to index.html at data-location=content
  index.addComponent(layout, "content");
  setContent(index); // set index.html as the root of our app
}
```

Listing 1: Initializing ProTeMs user interface inside `ProTeMUI.java`.

defined in `ProTeMUI.java`. Inside `processCommand()` the input string from the command line is passed on to a parser (`InstructionParser` inside the `scala` folder) which tries to determine whether the command is a valid assignment or computation, or if it contains invalid syntax. Valid commands are defined inside `Command.scala` where each possible command has its own case class extending the abstract class `Command` and overriding the mandatory method `execute()`. Listing 2 shows an excerpt of the defined commands. If the parser detected a valid command it returns an appropriate instance of one of those case classes, in case the command could not be parsed, it returns an exception. Valid commands can then be executed inside `processCommand()` and the appearance of the relevant UI components is updated accordingly. By employing AJAX for UI updates, changing the appearance of a Vaadin component does not result in a full reload of the webpage. Instead only relevant components are updated. This makes interactions with the app appear smoother and faster.

5 Conclusion

In this report we presented ProTeM, a tool that supports operations on proof terms that represent multisteps in first-order left-linear rewrite systems. We described the

```
case class Lhs(arg: terms.Rule) extends Command {
  override def execute(): Term = arg.lhs
}
case class CoInitial(arg1: Proofterm, arg2: Proofterm)
extends Command {
  override def execute(): Boolean = arg1.isCoInitial(arg2)
}
case class Measure(arg1: Proofterm, arg2: Proofterm)
extends Command {
  override def execute(): Any = {
    if(arg1.isCoInitial(arg2)) arg1.measure_overlap(arg2)
    else new NotCoinitial // return an Exception
  }
}
```

Listing 2: A few example commands as defined inside `Command.scala`.

operations ProTeM supports and provided some details about its implementation.

There are several possibilities to extend the functionality of ProTeM. First of all, adding a composition operation to the language of proof terms allows to represent rewrite sequences that are not single multisteps. Equivalence testing and normalization become then interesting questions. Also one could ask the tool to compute proof terms that represent a given rewrite sequence. Another useful extension will be automatic support for visualizing co-initial proof terms, like the figure in Example 2.1. Dropping the left-linearity requirement will be a challenging task, which requires the development of new theory.

References

[1] H. S. Bruggink. *Equivalence of Reductions in Higher-Order Rewriting.* PhD thesis, Utrecht University, 1980.

[2] N. Hirokawa and A. Middeldorp. Decreasing diagrams and relative termination. *Journal of Automated Reasoning*, 47(4):481–501, 2011.

[3] N. Hirokawa and A. Middeldorp. Commutation via relative termination. In *Proc. 2nd International Workshop on Confluence*, pages 29–33, 2013.

[4] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.

[5] Eclipse jetty. `https://www.eclipse.org/jetty/`. Accessed: 2018-02-01.

[6] C. Lombardi, A. Ríos, and R. de Vrijer. Proof terms for infinitary rewriting. In *Proceedings of the Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications*, volume 8560 of *Lecture Notes in Computer Science (Advanced Research in Computing and Software Science)*, pages 303–318, 2014.

[7] C. Marché, A. Rubio, and H. Zantema. Termination problem data base: Format of input files. `https://www.lri.fr/~marche/tpdb/format.html`. Accessed: 2018-01-17.

[8] Apache maven. `https://maven.apache.org/`. Accessed: 2018-02-02.

[9] J. Nagele. *Mechanizing Confluence.* PhD thesis, University of Innsbruck, 2017.

[10] J. Nagele and A. Middeldorp. Certification of classical confluence results for left-linear term rewrite systems. In *Proc. 7th International Conference on Interactive Theorem Proving*, volume 9807 of *Lecture Notes in Computer Science*, pages 290–306, 2016.

[11] Sass. `https://vaadin.com/docs/v8/framework/themes/themes-sass.html`. Accessed: 2018-02-02.

[12] scala-maven-plugin. `https://mvnrepository.com/artifact/net.alchim31.maven/scala-maven-plugin`. Accessed: 2017-12-28.

[13] Terese, editor. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 2003.

[14] Vaadin framework 8. `https://vaadin.com/docs/v8/framework/introduction/intro-overview.html`. Accessed: 2018-01-17.

[15] Vaadin archetype application. `https://mvnrepository.com/artifact/com.vaadin/vaadin-archetype-application`. Accessed: 2017-12-28.

[16] V. van Oostrom. Developing developments. *Theoretical Computer Science*, 175(1):159–181, 1997.

[17] V. van Oostrom and R. de Vrijer. Four equivalent equivalences of reductions. In *Proc. 2nd International Workshop on Reduction Strategies in Rewriting and Programming*, volume 70(6) of *Electronic Notes in Theoretical Computer Science*, pages 21–61, 2002.