Logical Semantics for the Rewriting Calculus

Aaron Stump

Dept. of Computer Science and Engineering, Washington University in St. Louis, St. Louis, MO, USA, Web: http://www.cse.wustl.edu/~stump

Carsten Schürmann

Dept. of Computer Science, Yale University, New Haven, CT, USA, Web: http://cs-www.cs.yale.edu/homes/carsten/

Abstract

The Rewriting Calculus has been proposed as a language for defining term rewriting strategies. Rules are explicitly represented as terms, and are applied explicitly to other terms to transform them. Sets of rules may be applied to (sets of) terms non-deterministically to obtain sets of results. Strategies are implemented as rules which accept other rules as arguments and apply them in certain ways. This paper describes work in progress to strengthen the Rewriting Calculus by giving it a logical semantics. Such a semantics can provide crucial guidance for studying the language and increasing its expressive power. The latter is demonstrated by adding support to the Rewriting Calculus for what we call *higher-form rewriting*, where rules rewrite other rules. The logical semantics used is based on ordered linear logic. The paper develops the ideas through several examples.

Key words: Rewriting Calculus, Ordered Linear Logic, Logical Semantics, Meta-Programming

1 Introduction

The untyped Rewriting Calculus of H. Cirstea and C. Kirchner is a higher-order functional language of explicit rewriting [3,4]. In contrast with standard term rewriting (see, e.g., [1]), the rewriting is explicit is the sense that rewrite rules must be applied explicitly to terms to transform them. For example, in order to transform a term $\neg\neg p$ using the rule $\neg\neg X \to X$, we must evaluate the expression

$$(\neg\neg X \to X) \ @ \ \neg\neg p$$

where we write @ for explicit application of a rule. The language is higher-order in the sense that rules and even sets of rules may be passed as arguments

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

to other rules. This enables parameterized strategies to be written [6]. A very simple example taken from the cited work is

$$u \to v \to x \to v \mathbin{@} (u \mathbin{@} x)$$

This term (call it *seq*) represents the strategy of rule sequencing: rules u and v are accepted, and a new rule is returned which will transform any term x by first applying u and then applying v to the result. So evaluating $\mathit{seq} \mathbin{@} (f(x) \to g(x)) \mathbin{@} (g(x) \to h(x)) \mathbin{@} f(a)$ yields $h(a)$.

The present work aims to strengthen the Rewriting Calculus in two ways. First, a definitive definition of the Rewriting Calculus has been elusive. For example, different versions of the language have been proposed in several papers (e.g. in [3] and [5]). One example of a design choice where it is hard to see which approach to choose has to do with applications involving sets. If a set of rules $\{R_1, \ldots, R_n\}$ is applied to a term T, then the result is defined to be $\{R_1 \mathbin{@} T, \ldots, R_n \mathbin{@} T\}$. This distributivity from the right of application over set formation is in accord with the idea that applying a set of rules to a term can have different results. Those results might not, for example, be joinable (if the rules are not confluent) or even semantically equivalent (e.g., if the rules are $a \to 0$ and $a \to 1$). The Rewriting Calculus just collects all the results in a set. So distributivity of application from the right seems reasonable. But what about distributivity from the left? If a single rule R is applied to a set of terms $\{T_1, \ldots, T_n\}$, should this evaluate to $\{R \mathbin{@} T_1, \ldots, R \mathbin{@} T_n\}$? It seems plausible, and would be very convenient to transform sets of results. But consider the case where the rule (call it R) is something like $\{x, y\} \to x + y$, with x and y variables. If we apply this rule to the set $\{2, 3\}$, we might expect to get 5. If we distribute the application over the target set $\{2, 3\}$, however, we will end up with $\{R \mathbin{@} 2,\ R \mathbin{@} 3\}$. In the version of the Rewriting Calculus from [3] (but not in the one from [5]), the expression $\{x, y\} \to x + y$ can evaluate to $\{x \to x + y,\ y \to x + y\}$. So we end up with the rather surprising $\{2 + y, x + 3, 3 + y, x + 2\}$, instead of 5. In the absence of another semantics for Rewriting Calculus terms, it is difficult to justify design decisions about matters like which kinds of distributivity to include in the operational semantics.

The present works also aims to strengthen the Rewriting Calculus by studying how to support what we call *higher-form* (to distinguish it from higher-order) rewriting. Higher-form rewriting enables rules to rewrite rules. A simple if somewhat artificial example is that of a rule which just reverses the left and right hand sides of a rule. So $f(x) \to g(x)$ would be rewritten to its reverse, $g(x) \to f(x)$. A natural way to try to write rule reversal would be as something like $(x \to y) \to (y \to x)$. The problem that quickly becomes apparent, however, is in dealing with the pattern variables of rules. In the Rewriting Calculus, the arrow is considered to be a binding construct [3]: all variables free in the left hand side (lhs) are considered bound by the arrow and subject to instantiation during pattern matching. As usual, terms are considered equivalent up to renaming of bound variables. So the term $(x \to y) \to (y \to x)$ is equivalent to $(x \to y) \to (z \to x)$, because the second occurrence of y is bound by the last arrow. And clearly this latter rule will not have the desired operational effect.

One attempt to improve this situation was made in [2], where arrow expressions are allowed to carry with them (typed) contexts declaring which variables are bound by the arrow and which are free. So we can write a rule like $(x \to_\emptyset y) \to_{\{x,y\}} (y \to_\emptyset x)$ where the subscripts of $\emptyset$ on the two arrows lower in the term show that the variables x and y are not bound by them; the $\{x, y\}$ subscript on the topmost arrow shows that x and y are bound there. But on the account given in [2], this rule is sufficient for reversing only ground rules like $f(a, b) \to_\emptyset g(b)$. We cannot use the proposed rule to reverse a rule with its own non-empty pattern variable context like $f(x, y) \to_{\{x,y\}} g(y)$, because the proposed rule requires $\emptyset$ for the context of the rule to be reversed. While it is possible to come up with an ad hoc operational semantics to allow higher-form rewriting, it again becomes difficult to justify certain design choices which arise.

The solution proposed here to these problems is to give what we call a *logical semantics* for the Rewriting Calculus. This semantics explains the meaning of constructs from term rewriting by interpreting them as logical constructs. In particular, we will interpret terms as logical formulas, and rewriting as logical entailment. Logic is a good place to look for a semantics for the Rewriting Calculus, given the origins of term rewriting in equational logic. Connecting the Rewriting Calculus to logic in a deep way can help guide the design of the language. We begin with a logical semantics for standard first-order term rewriting (Section 2). This semantics generalizes, with one modification, to interpret higher-form rewriting (Section 3). Finally (Section 4), we define the operational semantics of the Rewriting Calculus using our logical semantics, and compare it to the operational semantics given in [3]. Our semantics requires several seemingly minor extensions to well-understood logics. These extensions have not been satisfactorily studied yet, and so this remains work in progress.

2 Logical Semantics for First-Order Term Rewriting

We begin by giving a logical semantics for traditional unsorted first-order term rewriting. The intuition for our logical semantics has several ingredients. First, we note that performing a single step of rewriting involves using exactly one rule exactly once. The linearity restrictions suggest a linear logic. Second, if we wish to give a logical interpretation to semantically non-confluent rule sets like $\{a \to 0,\ a \to 1\}$, we can hardly interpret $\to$ as equality and set formation as conjunction. For then we would be rewriting with something whose logical interpretation was equivalent to false. Since false implies anything, all rewrites would be logically justified using such a rule set. It makes

$$\frac{}{A \vdash A}(Ax) \qquad \frac{}{\Gamma \vdash \top}(\top I) \qquad (1)\ \frac{\Gamma \vdash A \multimap B}{\Gamma \vdash p[A] \multimap p[B]}(Congr)$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}(\multimap I) \qquad \frac{\Gamma \vdash A \multimap B \qquad \Gamma' \vdash A}{\Gamma, \Gamma' \vdash B}(\multimap E)$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \& B}(\& I) \qquad (2)\ \frac{\Gamma \vdash A_1 \& A_2}{\Gamma \vdash A_i}(\& E)$$

$$(3)\ \frac{\Gamma \vdash [y/x]A}{\Gamma \vdash \forall x : O.A}(\forall I) \qquad \frac{\Gamma \vdash \forall x : O.A}{\Gamma \vdash [t/x]A}(\forall E)$$

(1) p an atomic context, (2) $i \in \{1, 2\}$, (3) y not free in Γ

Fig. 1. Our fragment of linear logic + (Congr)

more semantic sense to view a rule set as a resource which can be specialized exactly once to any of its rules. Finally, of course, syntactic pattern matching naturally suggests instantiation of universal quantifiers.

To give a semantics for term rewriting, we first adopt a definition of it (cf. [1,8]). We have a single sort I for all terms, together with a finite signature Σ of function symbols, each with a fixed arity. Terms are built from Σ and a countable set of variables in the usual way. The set of free variables $\mathrm{FV}(X)$ of a term X is defined as usual, as is the set of positions of a term. A rewrite rule is a pair of terms, written $L \to R$, where we require L not to be a variable and $\mathrm{FV}(R) \subset \mathrm{FV}(L)$. We extend FV to rules and sets of rules in the obvious way. We then say that term $\sigma(L)$ rewrites to $\sigma(R)$ at the top-level position using rule $L \to R$ iff σ is a substitution with domain including $\mathrm{FV}(L)$. We state that $f(t_1, \ldots, t_n)$ rewrites to $f(t'_1, \ldots, t'_n)$ at position $i \cdot \pi$ using such a rule iff t_i rewrites to t'_i at position π using that rule, and $t_j \equiv t'_j$ for all $j \neq i$. We also state that t rewrites to t' using a rule iff it does so at some position. Then t rewrites to t' using a finite set of rules $\mathcal{R}$ (notation: $t \Rightarrow_{\mathcal{R}} t'$) iff t rewrites to t' using one of the rules from $\mathcal{R}$. Finally, if t rewrites to each of $t_1, \ldots, t_n$ using $\mathcal{R}$, we write $t \Rightarrow_{\mathcal{R}} \{t_1, \ldots, t_n\}$. Notice that as a degenerate case of this last stipulation, we have $t \Rightarrow_{\mathcal{R}} \emptyset$ for all t and $\mathcal{R}$.

We now define our interpretation of terms, rules, and sets of rules. To model first-order terms as resources in the sense of linear logic, we must inter-

$$
\begin{aligned}
[\![x]\!] &:= x, \quad \text{for } x \text{ a variable} \\
[\![f(t_1,\ldots,t_n)]\!] &:= f[\![t_1]\!]\ldots[\![t_n]\!] \\
[\![t_1 \to t_2]\!] &:= \forall x_1 : O.\ldots\forall x_n : O.[\![t_1]\!]\multimap[\![t_2]\!], \\
&\quad\ \text{where } \{x_1,\ldots x_n\} = \mathrm{FV}(t_1 \to t_2) \\
[\![\{t_1,\ldots,t_n\}]\!] &:= [\![t_1]\!]\&\ldots\&[\![t_n]\!] \\
[\![t \Rightarrow_{\mathcal{R}} t']\!] &:= [\![\mathcal{R}]\!] \vdash [\![t]\!]\multimap[\![t']\!]
\end{aligned}
$$

Fig. 2. Interpretation of rewriting into linear logic

pret them as formulas. So our interpretation $[\![I]\!]$ of the single sort I of terms is O, the type of propositions. This forces us to model function symbols as higher-order (because taking in propositions) predicate symbols; we will view them as curried. We use $\rightharpoonup$ as the symbol for function space constructor, to reserve other arrow notations for more central concepts. The higher-order extension of our *base fragment* of standard intuitionistic linear logic (for what follows, we rely on [12]) is given in Figure 1. In that Figure, Γ is interpreted as a multiset of formulas. Our base fragment has just the operators $\forall$, $\multimap$, $\&$, and $\top$. We need additionally the congruence rule (Congr) so that we can rewrite at positions other than the top one. The side condition on (Congr) is that p is an *atomic context*, which we define to be being a ground term without connectives (i.e., a term built just from application and symbols from our signature) containing a single hole. A rule similar to (Congr) is admissible in our base fragment, but we lack further justification for (Congr) at present. Our interpretation is now given by Figure 2. The intention is for the rewriting judgment $t \Rightarrow_{\mathcal{R}} X$ to hold, where X is a term or set of terms, iff its interpretation is derivable.

For the interpretation of sets of terms, we understand the following degenerate cases by convention (the last one because $\top$ is the unit for $\&$):

$$
\begin{aligned}
[\![\{t\}]\!] &:= [\![t]\!] \\
[\![\emptyset]\!] &:= \top
\end{aligned}
$$

Example: Suppose $\mathcal{R}$ is $a \to b$. Then the rewriting judgment $f(a,c) \Rightarrow_{\mathcal{R}} f(b,c)$ holds, and its interpretation is derivable (where we take atomic context $(f \,\_\, c)$ in the use of (Congr)):

$$
\dfrac{\overline{a\multimap b \vdash a\multimap b}\,(Ax)}{a\multimap b \vdash (f\ a\ c) \multimap (f\ b\ c)}\,(Congr)
$$

Example: Suppose $\mathcal{R}$ is $\{f(x,g(y)) \to h(y), f(g(y),x) \to h(b)\}$. Then the rewriting judgment $f(g(a),g(a)) \Rightarrow_{\mathcal{R}} \{h(a),h(b)\}$ clearly holds. The corresponding sequent is easily observed to be derivable:

$$\begin{array}{ll}(\forall x : O.\forall y : O.f\ x\ (g\ y) \multimap (h\ y))\ \& & \\ (\forall x : O.\forall y : O.f\ (g\ y)\ x \multimap (h\ b)) & \vdash\ f\ (g\ a)\ (g\ a) \multimap (h\ a)\ \&\ (h\ b)\end{array}$$

We now prove completeness and restricted soundness of our semantics with respect to first-order rewriting. Note that we are viewing first-order rewriting as "ground truth" and our semantics as an attempt to express that ground truth in another way. Thus, it is correct to speak of completeness as the property that if a rewriting judgment holds, its interpretation is derivable (if it is true, then we can express it); and of soundness as the property that if the interpretation of a rewriting judgment is derivable, the judgment holds (if we express it, then it is true).

Theorem 1 (Completeness) *If the rewriting judgment $t \Rightarrow_{\mathcal{R}} X$ holds, where X is a term or set of terms, then its interpretation $[\![\mathcal{R}]\!] \vdash [\![t]\!] \multimap [\![X]\!]$ is derivable.*

Proof. By induction on the derivation of the rewriting judgment. If the rewriting step combines several results into a set of results, we use our induction hypothesis and the appropriate number of uses of $(\& I)$. If the rewriting step combines 0 results, then we use $(\top I)$. If the rewriting step uses rule $(L \to R) \in \mathcal{R}$, the induction hypothesis gives us (writing $\forall^*$ for universal closure of a formula) $\forall^* [\![L]\!] \multimap [\![R]\!] \vdash [\![t]\!] \multimap [\![X]\!]$. The result then easily follows from the elementary fact that $\mathcal{R} \vdash \forall^* [\![L]\!] \multimap [\![R]\!]$. If the rewriting step applies a rule at position π other than the top-level one, the induction hypothesis and (Congr) give us what we need. If the rewriting step is at the top-level, we instantiate the universal quantifiers for the interpretation of the rule in accordance with the matching substitution. □

Theorem 2 (Restricted Soundness) *If $[\![\mathcal{R}]\!] \vdash [\![t]\!] \multimap [\![t']\!]$ is derivable in such a way that (Congr) is used at most once, at the very end of the derivation; then $t \Rightarrow_{\mathcal{R}} t'$ holds.*

Proof. We may assume a derivation of $[\![\mathcal{R}]\!] \vdash [\![t]\!] \multimap [\![t']\!]$ without (Congr) is normal, in the usual proof-theoretic sense. Elimination rules may be applied to the axiom $[\![\mathcal{R}]\!] \vdash [\![\mathcal{R}]\!]$ to obtain a sequent of the appropriate form. No introduction rules then apply to give a sequent in the appropriate form. Applying elimination rules corresponds to choosing a rewrite rule and instantiating it. This corresponds to rewriting t to t' at the top level. Applying (Congr) corresponds to rewriting at a position in a surrounding term. □

Whether or not soundness holds without the restriction on the use of (Congr), in the next Section we cast off all restrictions on the uses of the rules to obtain a more general notion of rewriting. So of course, we cannot rely on our (restricted) soundness theorem in what follows. But that is indeed what we want. We have motived our semantics by showing that it is complete and sound (in a restricted way) for a well-understood form of rewriting. We then generalize the semantics in order to guide the definition of higher-form rewriting.

3 Logical Semantics for Higher-Form Rewriting

We can develop the linear logical semantics of the previous Section to support a much more general kind of rewriting than traditional first-order rewriting. We can support *higher-form* rewriting, where rules can rewrite other rules. This paradigm has the potential to enable rewriting meta-programs like completion or termination analysis to be written in the same language as the rewriting system itself, without reflection (cf. [7]).

To define higher-form rewriting, we first allow arbitrary signatures built from type O using $\rightharpoonup$. Now that we are leaving the first-order case, we need to use λ-abstractions, as is standard in higher-order logic (although this feature needs to be studied in combination with linearity). We need to make one refinement to the logic of Figure 1 to get a sensible notion of rewriting. We must interpret Γ as a list of formulas, rather than a multiset. This is necessary, for otherwise, if $\mathcal{R}$ is the ground rule $p \rightarrow q \rightarrow r$, then our semantics will say that q rewrites to $p \rightarrow r$ using $\mathcal{R}$. This is because the sequent

$$p \multimap q \multimap r \vdash q \multimap (p \multimap r)$$

is derivable in linear logic. But its derivation makes essential use of the ability to exchange the order of assumptions in the context. If we disallow this, the system given in Figure 1 becomes a fragment of the ordered linear logic of Polakow [11], if we interpret $\multimap$ as his ordered $\twoheadrightarrow$ connective (and retain the interpretation of & as a linear but not ordered connective). In ordered linear logic, the order in which hypotheses are used matters. In our fragment, hypotheses must be consumed (using $(\multimap E)$) in the order they were introduced. An example from [11] which helps get a feel for ordered linearity is the following. Using the rules of Figure 1 where Γ is interpreted as a list, we cannot derive $\vdash p \multimap (p \multimap q) \multimap q$. Any normal derivation (and these are sufficient for ordered linear logic [11]) would have to end with the following:

$$\dfrac{\dfrac{p, (p \multimap q) \vdash q}{p \vdash (p \multimap q) \multimap q}(\multimap I)}{\vdash p \multimap (p \multimap q) \multimap q}(\multimap I)$$

The obvious step to add above this derivation would be a use of $(\multimap E)$. But the only possible one for which we could finish the derivation would be:

$$\frac{p \multimap q \vdash p \multimap q \qquad p \vdash p}{p \multimap q, p \vdash q} (\multimap E)$$

And this has the assumptions in the context in its conclusion in the wrong order.

3.1 Narrowing Versus Rewriting

Just as in the previous Section, we define $t \Rightarrow_{\mathcal{R}} t'$ to hold iff its interpretation $[\![\mathcal{R}]\!] \vdash [\![t]\!] \multimap [\![t']\!]$ is derivable in our logic. We then define the multi-step rewriting relation $\Rightarrow^*_{\mathcal{R}}$ simply as the reflexive transitive closure of $\Rightarrow_{\mathcal{R}}$. We adopt the notation of our logic as the language of rewriting, enabling us to dispense with the uses of $[\![\cdot]\!]$. One important consequence of this definition must be pointed out. Our definition actually supports narrowing, not just rewriting. For example, suppose our single rule is $\forall x : O.f(h(x)) \multimap g(h(x))$. Then we can readily show that the term $\forall x : O.f(x)$ rewrites to $\forall x : O.g(h(x))$, since $\forall x : O.f(x) \vdash \forall x : O.f(h(x))$ is derivable.

In addition to the traditional kind of narrowing, our semantics supports another kind of narrowing, which we might call *result narrowing.* Sets of terms can be rewritten just by rewriting a proper subset. For example, our definition allows $f(a)$ & $f(b)$ & $g(c)$ to be rewritten to $g(a)$ & $g(b)$ using rewrite rule $\forall x : O.f(x) \multimap g(x)$. The fact that one term (here, $g(c)$) in a set of terms "gets stuck" (i.e., cannot be rewritten) does not cause the whole set of terms to get stuck. Result narrowing supports equational rewriting as it is done in the Rewriting Calculus. There, if an expression can be rewritten to more than one result using a given rule, due to the use of a non-unitary matching algorithm; then the set of all those results is returned. If some of those results cannot be rewritten later in evaluation, then they may just be dropped.

3.2 Provable Equivalence and Strongest Results

The rewriting relation we have defined needs some cleaning up. The operator & is associative, commutative, idempotent, and has unit $\top$. So equivalent sets of terms can be presented in many different ways. Furthermore, $(\forall I)$ can be used to introduce arbitrarily many trivial universal quantifiers, binding variables not free in the result. We refine our notion of rewriting by considering terms modulo provable equivalence. It should be possible to choose an intuitively sensible canonical representative for each induced equivalence class, but this must be explored in future work. Even with this refinement, we do not have a deterministic rewriting relation. For example, if $\mathcal{R}$ is $(a \multimap b)\ \&(a \multimap c)$, then $a \Rightarrow_{\mathcal{R}} b$ and $a \Rightarrow_{\mathcal{R}} c$. To obtain a deterministic relation, we can additionally stipulate that for $t \Rightarrow_{\mathcal{R}} t'$ to hold, t' must be *strongest*, in the sense that for any other t'' such $t \Rightarrow_{\mathcal{R}} t''$ holds, we have $\vdash t' \multimap t''$.

Signature:

$in : O \rightharpoonup O$	$* : O \rightharpoonup O \rightharpoonup O$ [*infix*]
$out : (O \rightharpoonup O) \rightharpoonup O$	$0 : O$
$build : O \rightharpoonup O$	$S : O \rightharpoonup O$
$exp : O \rightharpoonup O$	

Rules $\mathcal{R}$:

$(in(0) \multimap out(\forall x : O.exp\ x \multimap S(0)))\ \ \&$

$(\forall n : O.in(S(n)) \multimap build(in(n)))\ \ \&$

$(\forall u : O \rightharpoonup O.build(out(\forall x : O.exp\ x \multimap u(x))) \multimap out(\forall x : O.exp\ x \multimap x * u(x)))$

Fig. 3. Code Generation for Exponentiation

We spend the rest of this Section exploring our higher-form rewriting relation. For readability, we sometimes write applications like $(X\ Y)$ and sometimes like $X(Y)$. We associate $\multimap$ and & to the right, and application to the left.

3.3 *Example: Code Generation for Exponentiation*

Consider the problem of generating, from a number y, the rule $exp\ x \rightarrow x * \ldots * x * 1$, where the rhs has y occurrences of x. This is a simple meta-programming example taken from [9]. Note that it is quite different from the problem of simply computing x raised to the power y. We can solve this meta-programming problem quite elegantly in our new higher-form version of term rewriting. The signature and rules we need are given in Figure 3.

Proposition 1 *Suppose n is built just from 0 and S. Then we have $in(n) \Rightarrow^*_{\mathcal{R}} out(\forall x : O.exp\ x \multimap x * \ldots * x * S(0))$, where $x * \ldots * x * S(0)$ has n occurrences of x.*

Proof. The proof is by induction on the structure of n. If n is 0, we need, of course, just:

$$\cfrac{\overline{\mathcal{R} \vdash \mathcal{R}}^{(Ax)}}{\mathcal{R} \vdash in(n) \multimap out(\forall x : O.exp\ x \multimap S(0))}(\&\mathrm{E})$$

If n is $S(n')$, we first have

$$\dfrac{\dfrac{\overline{\mathcal{R} \vdash \mathcal{R}}(Ax)}{\mathcal{R} \vdash \forall n : O.in(S(n)) \multimap build(in(n))}(\&\text{E})\ \textit{twice}}{\mathcal{R} \vdash in(S(n'))\multimap build(in(n'))}(\forall E)$$

So we have $in(S(n')) \Rightarrow_{\mathcal{R}} build(in(n'))$. By our induction hypothesis, we have derivations for each step in $in(n') \Rightarrow^*_{\mathcal{R}} out(\forall x : O.exp\ x \multimap x * \ldots * x * S(0))$, where there are n' copies of x in the $\ldots$ expression. We extend each of those derivations using (Congr) to get derivations for each step of

$$build(in(n')) \Rightarrow^*_{\mathcal{R}} build(out(\forall x : O.exp\ x \multimap x * \ldots * x * S(0)))$$

Finally, we have

$$\dfrac{\dfrac{\overline{\mathcal{R} \vdash \mathcal{R}}(Ax)}{\begin{array}{c}\mathcal{R} \vdash \forall u : O \rightharpoonup O.build(out(\forall x : O.exp\ x \multimap u(x))) \\ \multimap\ out(\forall x : O.exp\ x \multimap x * u(x))\end{array}}(\&\text{E})\ \textit{twice}}{\begin{array}{c}\mathcal{R} \vdash build(out(\forall x : O.exp\ x \multimap x * \ldots * x * S(0))) \\ \multimap\ out(\forall x : O.exp\ x \multimap x * (x * \ldots * x * S(0)))\end{array}}(\forall E)$$

The term with which we have instantiated u in the use of $(\forall E)$ is $\lambda x : O.x * \ldots * x * S(0)$. We take our formulas to be β-normal forms, and so carry out the β-reduction tacitly in the derivation. By the definition of the reflexive transitive closure of $\Rightarrow_{\mathcal{R}}$, the rewritings we have demonstrated can now be combined to get the desired result. □

3.4 Example: Counting Bound Variables in λ-Calculus

Figure 4 defines an encoding function mapping terms of untyped λ-calculus into our rewriting language. The encoding uses *higher-order abstract syntax*, a well-known representation technique where bound variables of the object language (here, untyped λ-calculus) are represented as bound variables of the meta-language (here, our rewriting language) [10]. Here, the binding is accomplished using ∀. The last clause of the definition is for a single constant symbol c; we need just one for a representative example. Every λ-calculus term gets mapped to a formula (but not vice versa).

Figure 5 gives a signature and (numbered) rules for counting the number of occurrences of bound variables in a λ-calculus term. Standard rewrite

$$\underline{\text{Signature:}}\quad lam : O \rightharpoonup O, \quad app : O \rightharpoonup O \rightharpoonup O$$

$$\begin{aligned}
[\![\lambda x.M]\!] &:= lam\ (\forall x : O.[\![M]\!]) \\
[\![M\ \ N]\!] &:= app\ [\![M]\!]\ [\![N]\!] \\
[\![x]\!] &:= x, \quad \text{for } x \text{ a variable} \\
[\![c]\!] &:= c
\end{aligned}$$

Fig. 4. Encoding of untyped λ-calculus

$$\underline{\text{Signature:}}\quad num : O \rightharpoonup O, \quad + : O \rightharpoonup O \rightharpoonup O, \quad S : O \rightharpoonup O, \quad 0 : O$$

1. $\forall u : O \rightharpoonup O.num(lam\ (\forall x : O.u(x))) \multimap \forall x : O.num(u(x))$
2. $\forall M : O.\forall N : O.num(app\ M\ \ N) \multimap num(M) + num(N)$
3. $\forall M : O \rightharpoonup O.\forall N : O \rightharpoonup O.(\forall x : O.M(x) + N(x)) \multimap ((\forall x : O.M(x)) + (\forall x : O.N(x)))$
4. $(\forall x : O.num(x)) \multimap S(0)$
5. $num(c) \multimap 0$

Fig. 5. Rules to count bound variable occurrences in λ-calculus terms

rules for addition are not shown, and we abbreviate unary numerals using decimal. Note that this sort of computation cannot usually be implemented in a rewriting or functional programming language without using reflection. An example is given in Figure 6, where a sequence of rewritings to count the number of occurrences of bound variables in the encoding of the term $\lambda x.\lambda y.((c\ x)\ x)$ is shown. Each rewriting step is labeled with the number of the rule used. Several steps implicitly use the fact the if $x \notin \mathrm{FV}(M)$, then $\forall x : O.M \multimapboth M$.

3.5 Example: Congruence and &

Suppose our rules are $\mathcal{R} \equiv (f(a) \multimap d)\ \&\ (f(b) \multimap e)$. We have $\mathcal{R} \vdash f(a \& b) \multimap d \& e$; one half of the derivation is shown in Figure 7. So $f(a\&b) \Rightarrow_{\mathcal{R}} d\&e$ holds. On the other hand, if we let $\mathcal{R}' \equiv (d \multimap f(a))\ \&\ (e \multimap f(b))$, then we do not have $\mathcal{R}' \vdash d\&e \multimap f(a\&b)$. The reason is that while we can derive $\vdash p[a\&b] \multimap p[a]\&p[b]$ for atomic contexts p using the (Congr) rule, we cannot derive $\vdash p[a]\&p[b] \multimap p[a\&b]$.

$$\begin{array}{ll}
 & num(lam\ (\forall x : O.lam\ (\forall y : O.(app\ (app\ c\ x)\ x)))) \\
\Rightarrow_1 & \forall x : O.num(lam\ (\forall y : O.(app\ (app\ c\ x)\ x))) \\
\Rightarrow_1 & \forall x : O.num(app\ (app\ c\ x)\ x) \\
\Rightarrow_2 & \forall x : O.num(app\ c\ x)\ +\ num(x) \\
\Rightarrow_2 & \forall x : O.(num(c)\ +\ num(x))\ +\ num(x) \\
\Rightarrow_3 & (\forall x : O.num(c)\ +\ num(x))\ +\ (\forall x : O.num(x)) \\
\Rightarrow_4 & (\forall x : O.num(c)\ +\ num(x))\ +\ 1 \\
\Rightarrow_3 & ((\forall x : O.num(c))\ +\ (\forall x : O.num(x)))\ +\ 1 \\
\Rightarrow_4 & (num(c)\ +\ 1)\ +\ 1 \\
\Rightarrow_5 & (0\ +\ 1)\ +\ 1 \\
\Rightarrow_{arith} & 2
\end{array}$$

Fig. 6. Counting uses of bound variables in $\lambda x.\lambda y.((c\ x)\ x)$

$$\dfrac{\dfrac{\dfrac{}{f(a){\multimap}d \vdash f(a){\multimap}d} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{a\&b \vdash a\&b}}{a\&b \vdash a}(\&E)}{\vdash f(a\&b){\multimap}f(a)}(Congr) \qquad \dfrac{}{f(a\&b) \vdash f(a\&b)}}{f(a\&b) \vdash f(a)}({\multimap}E)}{f(a){\multimap}d, f(a\&b) \vdash d}({\multimap}E)}{f(a){\multimap}d \vdash f(a\&b){\multimap}d}({\multimap}\mathrm{I})$$

Fig. 7. Example derivation using congruence

4 The Rewriting Calculus

We use our logical semantics to define the Rewriting Calculus. We take Rewriting Calculus terms to be terms of the ordered linear logic we have been considering, with an additional construct @ for explicit rule application. The evaluation relation $t \Rightarrow_\rho t'$ of the Rewriting Calculus is defined to be the congruence closure of the relation stating how explicit rule applications $x@y$ are evaluated, where x and y are both of type O. Note that x is not of functional type, because we are modeling rewrite rules as formulas. For ground terms x and y, we stipulate that $x@y$ evaluates to z where $y \Rightarrow_x z$. To generalize this to terms with free variables, we require that $\sigma(y) \Rightarrow_{\sigma(x)} \sigma(z)$,

$$\begin{array}{lll}
(\mathit{Fire}) & (l \rightarrow r) \mathbin{@} t & \Longrightarrow \{\sigma_1(r), \ldots, \sigma_n(r)\} \\
(\mathit{Distrib}) & \{u_1, \ldots, u_n\} \mathbin{@} v & \Longrightarrow \{u_1 \mathbin{@} v, \ldots, u_n \mathbin{@} v\} \\
(\mathit{Batch}) & v \mathbin{@} \{u_1, \ldots, u_n\} & \Longrightarrow \{v \mathbin{@} u_1, \ldots, v \mathbin{@} u_n\} \\
(\mathit{Switch_L}) & \{u_1, \ldots, u_n\} \rightarrow v & \Longrightarrow \{u_1 \rightarrow v, \ldots, u_n \rightarrow v\} \\
(\mathit{Switch_R}) & v \rightarrow \{u_1, \ldots, u_n\} & \Longrightarrow \{v \rightarrow u_1, \ldots, v \rightarrow u_n\} \\
(\mathit{OpOnSet}) & f(v_1, \ldots, \{u_1, \ldots, u_m\}, \ldots, v_n) & \Longrightarrow \{f(v_1, \ldots, u_1, \ldots, v_n), \ldots, \\
& & \quad f(v_1, \ldots, u_m, \ldots, v_n)\} \\
(\mathit{Flat}) & \{v_1, \ldots, \{u_1, \ldots, u_m\}, \ldots, v_n\} & \Longrightarrow \{v_1, \ldots, u_1, \ldots, u_m, \ldots, v_n\}
\end{array}$$

where $\sigma_1, \ldots, \sigma_n$ are all substitutions satisfying $\sigma(l) \equiv t$

Fig. 8. Cirstea-Kirchner definition of the Rewriting Calculus

for all substitutions σ for those free variables. This should be achievable by requiring $\vdash \forall^* x \multimap y \multimap z$. Since in the Rewriting Calculus, rules are applied only at the top level of terms, we must restrict the use of the (Congr) rule of Figure 1 when establishing $y \Rightarrow_x z$. We require that the context Γ be empty for (Congr) to be applied. This rules out rewriting at positions other than the topmost one in terms, but it still allows us to derive $\vdash p[a\&b] \multimap p[a]\&p[b]$.

Our definition of the Rewriting Calculus's evaluation relation is to be contrasted with the definition given in [3, Section 2], as the congruence closure of the relation defined by the rules reproduced in Figure 8. Two rules which are redundant in the presence of the others are omitted. The rules are largely concerned with manipulation of sets of results (which are treated up to associativity, commutativity, and idempotence). Of the rules not involving @, we find that (*Switch*$_R$), (*Flat*) and (*OpOnSet*) are sound with respect to our semantics, in the sense that the interpretation of the lhs indeed logically entails the interpretation of the rhs. The lhs and rhs of (*Switch*$_R$) and (*Flat*) are, in fact, logically equivalent, although those of (*OpOnSet*) are not. Our semantics also agrees with (*Fire*), (*Distrib*) and (*Batch*). But (*Switch*$_L$) is not sound with respect to our semantics, since in general $u_1 \& \ldots \& u_n \multimap v$ does not imply (and is not implied by) $(u_1 \multimap v) \& \ldots \& (u_n \multimap v)$.

One detail of these observations is worth drawing out in more detail. The rules (except (*Switch*$_L$)) are sound with respect to our semantics even when the sets involved are empty. Recall that we interpret $\emptyset$ as $\top$. Note, further, that since $\Gamma \vdash \top$ for any context Γ, we always have $t \Rightarrow_{\mathcal{R}} \top$, for all t and $\mathcal{R}$. Consider now, say, the rule (*Batch*). When the set involved is empty, the interpretation of this rule is that $v \mathbin{@} \top$ rewrites to $\top$. Indeed, $\top$ is the strongest formula G such that $v \vdash \top \multimap G$ holds. Similarly, for the rule (*Distrib*), we have $\top \mathbin{@} v \multimap \top$, and $\top$ is again the strongest implied formula. This is a nice further confirmation that the logical semantics is appropriate for the Rewriting Calculus (minus ($Switch_L$)).

It would be quite interesting to determine whether or not the rules of Figure 8 are complete for our logical semantics (assuming just first-order instead of higher-form rewriting), or perhaps complete under some restrictions.

5 Conclusion

We have considered a logical semantics for traditional rewriting and also what we call higher-form rewriting, where rules may rewrite rules. The semantics is based on ordered linear logic, with some higher-order extensions. Terms are interpreted as formulas, and rewriting as ordered linear entailment. All but one operational rule of the Rewriting Calculus as given by Cirstea and Kirchner are sound with respect to the proposed semantics.

There is clearly much future work to be done. Although relatively minor, the proposed higher-order extensions to ordered linear logic must be studied. Extending the work to multi-sorted systems would seem to require different "flavors" of the propositional type O corresponding to different sorts, which is not standard. But the main goal is to devise an effective operational semantics which is sound and hopefully complete with respect to the proposed logical semantics. We would then have achieved our goal of a semantically motivated definition of the Rewriting Calculus which supports what promises to be a powerful meta-programming paradigm of higher-form rewriting.

Acknowledgments: Thanks to the anonymous reviewers for helpful comments on earlier drafts of this paper, and to Horatiu Cirstea, Germain Fauré, Claude Kirchner, Luigi Liquori, Benjamin Wack, and other members of the PROTHEO team for many excellent discussions on the Rewriting Calculus.

References

[1] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[2] G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure patterns type systems. In *Principles of Programming Languages*. ACM, 2003.

[3] H. Cirstea and C. Kirchner. The Rewriting Calculus - Part I. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9:363–399, May 2001. Also available as Technical Report A01-R-203, LORIA, Nancy (France).

[4] H. Cirstea and C. Kirchner. The Rewriting Calculus - Part II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9:401–434, May 2001. Also available as Technical Report A01-R-204, LORIA, Nancy (France).

[5] H. Cirstea, C. Kirchner, and L. Liquori. Matching Power. In A. Middeldorp, editor, *Proceedings of the 12th International Conference on Rewriting*

Techniques and Applications, LNCS, Utrecht (The Netherlands), 2001. Springer-Verlag.

[6] H. Cirstea, C. Kirchner, L. Liquori, and B. Wack. Rewrite strategies in the rewriting calculus. In B. Gramlich and S. Lucas, editors, *Proceedings of the Third International Workshop on Reduction Strategies in Rewriting and Programming* , Valencia, Spain, June 2003. Electronic Notes in Theoretical Computer Science.

[7] M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In *Cafe: An Industrial-Strength Algebraic Formal Method*. Elsevier, 2000.

[8] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[9] A. Nanevski. Meta-programming with names and necessity. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming*, pages 206–217, 2002.

[10] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *ACM SIGPLAN Symposium on Language Design and Implementation*, 1988.

[11] J. Polakow. *Ordered Linear Logic and Applications*. PhD thesis, Carnegie Mellon University, 2001.

[12] A. Troelstra. *Lectures on Linear Logic*. Center for the Study of Language and Information, 1992.