From Rogue to MicroRogue

Aaron Stump, Ryan Besand, James C. Brodman,
Jonathan Hseu, and Bill Kinnersley

Dept. of Computer Science and Engineering, Washington University in St. Louis, St. Louis, Missouri, USA, Web: http://cl.cse.wustl.edu/

Abstract

The Rewriting Calculus has been proposed as a foundational system combining the central ideas of λ-calculus and term rewriting. The rewriting is explicit, in the sense that rules must be applied explicitly to terms to transform them. This paper begins with an imperative version of the Rewriting Calculus called Rogue. It then shows how Rogue can itself be conveniently implemented by an even more foundational system called MicroRogue. MicroRogue rewrites terms using a global set of first-order rules. Rules can be enabled, disabled, and dynamically added in scopes, which can be pushed and popped. MicroRogue also provides mechanisms for specifying evaluation order. Using these primitives, a Rogue interpreter can be implemented in less than 40 lines of MicroRogue code.

Key words: Rewriting Calculus, dynamic rules, evaluation order

1 Introduction

The Rewriting Calculus has been proposed as a foundational system combining the central ideas of λ-calculus and term rewriting [6,7]. The rewriting is explicit, in the sense that rules must be applied explicitly to terms to transform them. This differs from traditional approaches to term rewriting, where a term is subject nondeterministically to transformation at any of its subexpressions by any of a given global set of rewrite rules (see, e.g., [4,9]). Indeed, one of the original motivations in the development of the Rewriting Calculus was to provide an operational semantics for systems implementing traditional term rewriting [6,7].

The starting point for this paper is an imperative version of the Rewriting Calculus called Rogue. Rogue has been used to implement decision procedures [12], as well as other symbolic programs including proof checkers, type checkers, and standard automata-manipulating algorithms for lexer and parser generation. In these applications, the code comes out much more concisely than implementations in more standard languages. Concision is valuable not

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

$$T ::= c \;\parallel\; x \;\parallel\; \texttt{null} \;\parallel\; T_1.T_2 \;\parallel\; T_1@T_2 \;\parallel\; T_1, T_2 \;\parallel$$
$$T_1 \to T_2 \;\parallel\; T_1 \Rightarrow T_2 \;\parallel\; T_1|T_2 \;\parallel\; T_1.T_2 := T_3 \;\parallel\; \alpha\, x.T$$

Fig. 1. The syntax of Rogue

just for aesthetic reasons: minimizing the trusted computing base is important for security applications (see, e.g., [2,3]).

The main concern of this work is to show how Rogue can itself be concisely implemented in a simpler rewriting system, called MicroRogue. MicroRogue resembles a traditional implementation of term rewriting insofar as it rewrites terms using a global set of rules. The rules are only applied at the top level of terms, however. Rules are ordered: if more than one applies, the greatest in the ordering is used. Even less like traditional term rewriting, rules can be dynamically enabled and disabled. Inspired by work of Visser, we also allow new rules to be added dynamically [13]. Rules are added in scopes, which can be pushed and popped. Finally, MicroRogue provides mechanisms for specifying evaluation order.

Using the primitives of MicroRogue, a Rogue interpreter can be implemented in less than 40 (non-comment, non-blank) lines of MicroRogue code. This is attractive from a practical point of view, since it enables much easier development of Rogue than seems possible in an industrial non-symbolic language like C++. From a more theoretical perspective, the move from Rogue to MicroRogue is motivated by the observation that the definition of the operational semantics of the Rewriting Calculus contains several clauses which are essentially simple rewriting rules. This suggests that a meta-language for implementing Rogue or the Rewriting Calculus should be based on some kind of rewriting. This seems somewhat strange, since these are already supposed to be foundational rewriting languages! MicroRogue resolves this tension by providing a small set of rewriting primitives which are sufficient to implement the operational semantics of Rogue.

The rest of the paper is organized as follows. Section 2 describes Rogue. Section 3 discusses the forces leading to the development of MicroRogue from Rogue. Section 4 defines MicroRogue, and Section 5 walks through the implementation of Rogue in MicroRogue.

2 Rogue

The Rogue programming language is essentially a version of the untyped Rewriting Calculus [6,7]. The crucial ideas in the Rewriting Calculus are to make application of rewrite rules explicit, and to make it possible to pass rewrite rules, and indeed sets of rewrite rules, around as first-class data. Rogue fixes a particular pattern-matching algorithm and evaluation strategy, which the definition of the Rewriting Calculus leaves, to a large extent, as customizable parameters. Rogue differs from the Rewriting Calculus in adding an explicit scoping operator for declaring variables, and adding mutable expression attributes and recursive definitions[1]. Recursive definitions are just for convenience, since the Rewriting Calculus is already Turing-complete without them. Figure 1 gives the syntax for Rogue. Operators are listed in order from tightest to loosest binding. All binary operators are right associative except "@" and ".", which are left associative. We write $\alpha\, x.\, E$ in ASCII as `x ^ E`, and allow $x(y)$ as alternative notation for x @ y when x is a variable or a constant.

2.1 Rogue Basics

A formal definition of Rogue's operational semantics will be given in the form of a MicroRogue program in Section 5. For now, the language is introduced through some basic examples. Expressions are generally evaluated in leftmost, innermost order, except that evaluation does not take place on the right hand sides of arrow expressions. Here are three one-step evaluations of applications of ground rules to terms:

```
1. (a -> b)       @ a    ===> b
2. (f(a) -> g(b)) @ f(a) ===> g(b)
3. (f(a) -> g(b)) @ f(c) ===> null
```

The first two applications succeed because the left hand side of the rule that is being applied is identical to the term to which the rule is applied. The third application fails (with `null`) since `f(a)` is not identical to `f(c)`. Note that as stated above, `f(a)` is just alternative notation for `f @ a`. Consider now the following one-step evaluations of applications of rules with variables:

```
1. (x^   x -> x)            @ a      ===> a
2. (x^y^ x,y -> y,x)        @ (a,b)  ===> (b,a)
3. (x^   f(x,x) -> g(x,x)) @ f(c,c) ===> g(c,c)
4. (x^   f(x,x) -> g(x,x)) @ f(c,d) ===> null
```

The first three rules succeed because the left hand sides of the rules being applied all match the target terms. For example, `f(x,x)` matches `f(c,c)` with the substitution $[c/x]$ ("replace x by c"), where the use of the scoping operator declares x to be a variable available for instantiation. But there is no matching substitution for pattern `f(x,x)` and target `f(c,d)`, since the pattern requires identical subexpressions, but `c` and `d` are syntactically distinct. Here are evaluations demonstrating the use of comma:

```
1. (x^   (a -> b), (x -> x)) @ a      ===>
         (a -> b) @ a, (x^x -> x) @ a ===> b,a
2. (     (a -> b), (c -> d)) @ a      ===>
         (a -> b) @ a, (c -> d) @ a   ===>
                           b,null   ===> b
```

[1] The explicit scoping operator is new and still being developed (see [1]).

We distribute comma expressions from the left (but in Rogue, not from the right) over application. So in the first example, we apply both rules to `a`, and collect the results in a comma expression. As the second example shows, if one of the rules (`c -> d`) fails to match, then the result is `(b,null)`, which reduces to just `b`. The bar operator is essentially the *first* operator of the Rewriting Calculus [7]. Applications of bar expressions are evaluated by applying the first (from left to right) rule that matches the target:

```
1. (x^   f(x) -> x | x -> a) @ f(b)    ===> b
2. (x^   f(x) -> x | x -> x) @ g(d)    ===> g(d)
```

Rogue also has a lazy arrow (`=>`), which is very useful for meta-programming and reflection. It allows unevaluated expressions to be manipulated (we include here arithmetic operations, which we omit from our formal presentations but which can easily be included):

```
1. (x^y^ x,y -> y)   @ (null,3)      ===>
   (x^y^ x,y -> y)   @ 3             ===> null
2. (x^y^ x,y => f(x),y+y) @ (null,3) ===>
                          f(null),3+3 ===> f(null),6
```

In the first example, `(null,3)` is first reduced to `3`, which does not match the pattern `(x,y)`. So the whole application reduces to `null`. But in the second example, because a lazy arrow expression is applied, we do not reduce `(null,3)`. This unevaluated expression then matches the pattern `(x,y)`, and evaluation continues successfully.

2.2 Programming Examples

This section gives some examples that are useful for programming in Rogue. First, we have definitions of `if-then-else` and `if`. These are defined using the lazy arrow ($\Rightarrow$), since the if-part must be evaluated before the other part(s). We have that `if(a,b)` evaluates to whatever `b` does if `a` evaluates to something non-`null`, and `null` otherwise.

```
ite := x^y^z^ x, y, z => (null -> z | q^q -> y) @ x
if := x^y^ x, y => ite(x,y,null)
```

We can also define boolean operations and equality; these definitions allow conjunctions and disjunctions with arbitrary numbers of conjuncts and disjuncts, respectively. Furthermore, if a conjunct is found in evaluation from left to right of all the conjuncts which evaluates to false, the conjuncts to the right of that one are not evaluated. For example, `and(null, (a := b))` evaluates to `null` without causing `a` to be set to `b`.

```
and := p^q^ p,q => ite(and(p),and(q),null) | p -> p ;;
not := p^   null -> 1;;
or  := p^q^ p,q => ite(or(p),1,or(q)) | p -> p ;;
eq  := x^   x,x -> 1;;
```

```
find := x^ x ->
 ite(x.findp,
     let(top^top, find(x.findp), (x.findp := top)),
     null(x.rank := 0), x);;
```

Fig. 2. Rogue code for find with path-compression

Two more definitions prove very useful. The first is a pattern-matching `let` statement. With this definition, `let(x,y,z)` evaluates to `null` if there is no substitution σ which makes pattern `x` match target expression `y`; and otherwise it evaluates to $\sigma(z)$.

```
let := x^y^z^ x,y,z => (x -> z) @ y;;
```

For example, `let(x^f(x),f(3),x+4)` evaluates to `7`. The final definition is for a function we call `apply` (otherwise known as `map`), which applies a function to every element of a comma tree. So `apply(x^x -> x+10) @ ((1,2),(3,4))` evaluates to `((11,12),(13,14))`.

```
apply := F^ F -> x^y^(x,y   -> apply(F) @ x, apply(F) @ y |
                      null  -> null | x -> F(x));;
```

2.3 A Richer Example

We consider a more involved example, taken from the application domain mentioned previously of decision procedures [12]. Figure 2 give Rogue code for the find function of the union-find data structure [8, Chapter 22]. The find function is supposed to follow find pointers, stored using the `findp` attribute, from an element `x` until it reaches a node without a find pointer. This latter node serves as the canonical representative for `x`'s equivalence class, and is returned by `find`. If `x` had a find pointer, it is modified to point to this `top` node. The Rogue code in the Figure takes in the element `x`. Then, depending on whether or not `x`'s find pointer `x.findp` is , it either makes a recursive call on `x.findp` or simply evaluates to `x`. If it makes the recursive call `find(x.findp)`, then it modifies `x.findp` subsequently. Recall that attribute assignments evaluate to whatever value is being assigned, so `x.findp := top` evaluates to `top`. In the other case where `x.findp` is `null`, the code first sets `x`'s *rank* to 0. This is for the benefit of the code for union (not shown). By applying `null` to that attribute assignment, we cause the value (0) to drop out, since `null` applied to anything is `null`. So we get `(null,x)` for the else-part of the if-then-else, which evaluates to `x`.

3 Towards MicroRogue

This section explains the various forces arising in implementing and using Rogue which lead us towards MicroRogue. Most of the features of MicroRogue come directly from an attempt to satisfy these design forces simultaneously.

3.1 Rewriting in Rogue's Operational Semantics

The definitions of Rogue and the Rewriting Calculus contain a number of clauses that look very much like rules from traditional term rewriting. For example, we have the following rules:

$$
\begin{aligned}
(r_1, r_2) \ @ \ t &\Longrightarrow (r_1 @ t),\ (r_2 @ t) \\
\texttt{null} \ @ \ t &\Longrightarrow \texttt{null} \\
x, \texttt{null} &\Longrightarrow x \\
\texttt{null}, x &\Longrightarrow x
\end{aligned}
$$

These can very naturally be viewed as term rewriting rules, which suggests that a good meta-language for implementing Rogue or the Rewriting Calculus would be some kind of rewriting language. This is true even though rewrite rules in the Rewriting Calculus do not map directly onto rewriting rules in standard term rewriting (for example, nonregular rules are allowed in the Rewriting Calculus). Since the Rewriting Calculus is itself proposed as a foundation for implementing rewriting systems, it seems in a sense somewhat paradoxical that the meta-language for describing it involves rewriting. That is, the foundation (Rewriting Calculus) seems to depend on that which it is intended to serve as a foundation for (term rewriting). To resolve this tension, we seek to accomodate:

Force 1 *Enable the operational semantics of Rogue to be written using rewrite rules for cases like* $(r_1, r_2) \ @ \ t \ \Longrightarrow \ (r_1 @ t),\ (r_2 @ t)$.

3.2 Definitions and Mutable Attributes

Simple definitions like the ones for `ite` and other the basic examples above also seem just like simple rewrite rules. The defined symbol should be replaced by the body of the definition whenever it is encountered during evaluation. Since definitions are added in the body of a Rogue program, as opposed to the definition of Rogue's operational semantics, some facility for dynamically adding rules (after the definition of the operational semantics of the language) would be needed. Looking up the value for an attribute E_2 of of an expression E_1 could also just be thought of as rewriting $E_1.E_2$ to its value. Naturally, since attributes can be set dynamically, it would have to be possible to add and modify rules dynamically to implement attribute assignments with rules. These considerations lead to another design force for MicroRogue:

Force 2 *Support definitions and mutable attributes using dynamic rules.*

3.3 Evaluation Order

If we are to implement Rogue's operational semantics in a more foundational language, that language must provide support for controlling the order of evaluation of Rogue expressions. Rogue's evaluation order is somewhat involved. Applications $M@N$ must be evaluated carefully, since if M evaluates to a lazy arrow expression, or a bar expression beginning with a lazy arrow expression, N should not be evaluated. Also, we need to be able to specify that no evaluation takes place in certain positions; namely, on the right hand side of arrow expressions. This leads to:

Force 3 *Enable fine-grained control of evaluation order.*

3.4 Backtrackable State

In implementing imperative decision procedures for quantifier-free formulas of first-order theories, it is important to be able to backtrack all state of the decision procedure on command [12,5]. This is because these decision procedures are integrated with propositional SAT solvers which perform backtracking search of the space of boolean assignments to the atomic subformulas of the goal. The SAT solver may decide to back out partially from some particular assignment, in which case the decision procedures must also backtrack their state to remain in synch with the SAT solver. So at least for these kinds of applications, it is highly useful to have:

Force 4 *Enable backtracking of dynamically added or modified rules.*

4 MicroRogue

MicroRogue is designed to satisfy the design forces of the preceding Section, to serve as a foundational rewriting language suitable for implementing Rogue and other higher-level rewriting languages. The set of MicroRogue expressions T is defined in Figure 3, with constructs given in order from tightest to loosest binding. The operators are nonassociative, except for semicolon and tilde semicolon, which are both right associative. A set of *basic expressions* E of interest (for example, the expressions of the Rogue input language) is assumed. These may be built over constant and variable symbols using some operators. We make use of a set of positions *Pos*, which are just finite sequences of natural numbers. They are used in controlling the evaluation order for different forms of expressions. Arrow expressions are for rules. They are either anonymous ($T_1 \rightarrow T_2$) or named ($i : T_1 \rightarrow T_2$, where i is take from some set of names). Scopes for variables are declared with α-expressions, as in Rogue. Semicolon and tilde semicolon are for right and left sequencing, respectively. In each kind of sequencing, the subterms are evaluated in left-to-right order. In right sequencing (";"), the whole expression evaluates to whatever the right subterm evaluates to. In left sequencing ("˜;"), it evaluates to whatever the left subterm evaluates to. We have constructs to turn rules on and off, as mentioned above. We turn now to the construct T !.

The operational semantics of MicroRogue defines how a certain kind of state is updated during evaluation of a MicroRogue expression. This state

$$
\begin{array}{lll}
T & ::= & E \;\|\; Pos \;\|\; T_1 \rightarrow T_2 \;\|\; i : T_1 \rightarrow T_2 \;\|\; \mathit{ON}\ T \;\| \\
 & & \mathit{OFF}\ T \;\|\; T\,! \;\|\; T_1 ; T_2 \;\|\; T_2\,\tilde{};T_2 \;\|\; \alpha\, x.T \;\|\; \mathit{PUSH} \;\| \\
 & & \mathit{POP} \;\|\; \mathit{ENDCORE} \;\|\; \mathit{DONE} \\
Pos & ::= & . \;\|\; .NumList \\
NumList & ::= & N \;\|\; N.NumList
\end{array}
$$

Fig. 3. The syntax of MicroRogue

has two parts. There is a global set of rules to use for top-level rewriting of expressions. There is also a single MicroRogue expression called the *hold* expression. The hold expression can have its subexpressions rewritten in place. This is how MicroRogue allows evaluation order to be controlled. Both the global set of rules and the hold expression are modified using the bang construct. When T ! is evaluated with T a rule, T is added to the global set of rules. When T ! is evaluated with T a position, then the hold expression H is modified as follows. Write $H|_T \downarrow$ to indicate that T is indeed a valid position into H. Then if $H|_T \downarrow$, the hold expression is modified to become $H[Q]_T$, where Q is what $H|_T$ evaluates to (following standard notation from, e.g., Chapter 3 of [4]). One other operation sets the hold expression: whenever we try to find a global rule applicable to expression X, the hold expression is set to be X.

Evaluating *DONE* causes the hold expression to be marked as normalized. This is useful, for example, to prevent congruence rules from being needlessly applied to a term which has already been fully evaluated. In order to accomodate dynamically added rules, however, expressions are marked as normalized just with respect to some core prefix of the global set of rules. Dynamic rules may still be used to rewrite expressions that have been normalized with respect to the core rules. The core prefix of rules is indicated by evaluating the special expression *ENDCORE*. If a term has been marked as normalized and the first rule in the ordering which applies to it is in the set of rules added before the evaluation of *ENDCORE*, then the term will be rewritten just to itself. Otherwise, the rewriting will proceed.

Formal evaluation rules are given in Figures 4 and 5. For typographic reasons, the names of the rules and their side conditions are written above the rules. The derivable objects are sequents $R :: H :: T \implies R' :: H' :: T'$, showing the starting rule list R, hold expression H, and current MicroRogue expression T to be rewritten; and the resulting rule list, hold expression, and result T' for T. Rule lists are lists of annotated rules $i : l \rightarrow^{?} r$, where i is the unique identifier for the rule and ? is either $+$ or $-$ indicating whether the rule is enabled or disabled. Rules are initially enabled when added. The rules in each list are (totally) ordered by the time they were added to that

(right-seq)

$$\frac{R :: H :: T_1 \implies R_1 :: H_1 :: T_1' \qquad R_1 :: H_1 :: T_2 \implies R_2 :: H_2 :: T_2'}{R :: H :: T_1 ; T_2 \implies R_2 :: H_2 :: T_2'}$$

(left-seq)

$$\frac{R :: H :: T_1 \implies R_1 :: H_1 :: T_1' \qquad R_1 :: H_1 :: T_2 \implies R_2 :: H_2 :: T_2'}{R :: H :: T_1 \tilde{;} T_2 \implies R_2 :: H_2 :: T_1'}$$

(on)

$$\frac{}{R,\ i : l \rightarrow^{?} r,\ R' :: H :: \mathit{ON}\, i \implies R,\ i : l \rightarrow^{+} r,\ R' :: H :: \texttt{null}}$$

(off)

$$\frac{}{R,\ i : l \rightarrow^{?} r,\ R' :: H :: \mathit{OFF}\, i \implies R,\ i : l \rightarrow^{-} r,\ R' :: H :: \texttt{null}}$$

(add-rule1) $\quad i$ fresh

$$\frac{}{R :: H :: l \rightarrow r\ ! \implies R,\ i : l \rightarrow^{+} r :: H :: i}$$

(add-rule2)

$$\frac{}{R :: H :: i : l \rightarrow r\ ! \implies R,\ i : l \rightarrow^{+} r :: H :: i}$$

Fig. 4. The operational semantics of MicroRogue (first part)

list. Scopes are separated in rule lists using $\diamond$. When a POP is evaluated, all rules added since the previous unpopped $PUSH$ are removed. Expressions E are marked as normalized by the (done) rule by writing E^n; we omit this notation from other rules. We write $R(T)$ for the result of applying rule list R to expression T, which is defined as follows. Let $l \rightarrow r$ be the first (from right to left) enabled rule of R such that l matches T. If T is unmarked or else this rule is not to the left of any $\blacktriangleleft$ (added by $ENDCORE$) in R, then

(push)

$$\frac{}{R :: H :: PUSH \implies R \diamond :: H :: \texttt{null}}$$

(pop) $\quad \diamond \notin R'$

$$\frac{}{R \ \diamond \ R' :: H :: POP \implies R :: H :: \texttt{null}}$$

(subexpr) $\quad H|_P \downarrow,\ H' \equiv H[Q]_P$

$$\frac{R :: H :: H|_P \implies R' :: H_1 :: Q}{R :: H :: P \ ! \implies R' :: H' :: H'}$$

(done)

$$\frac{}{R :: H :: DONE \implies R :: H :: H^n}$$

(endcore)

$$\frac{}{R :: H :: ENDCORE \implies R \blacktriangleleft :: H :: H}$$

(basic-1) $\quad T \equiv R(T)$, and no other rules apply

$$\frac{}{R :: H :: T \implies R :: T :: T}$$

(basic-2) $\quad T \not\equiv T' \equiv R(T)$, and no other rules apply

$$\frac{R :: T :: T' \implies R' :: H' :: Q}{R :: H :: T \implies R' :: H' :: Q}$$

Fig. 5. The operational semantics of MicroRogue (second part)

we define $R(T)$ to be $\sigma(r)$ where σ is the matching substitution for l and T. Otherwise, or if there is no matching rule at all, we define $R(T)$ to be just T.

The rules of Figures 4 and 5 are used to evaluate a MicroRogue expression starting with the empty rule list and empty hold expression. The rules are directly executable by applying them bottom-up as a logic program. The left hand side of the sequent is the input, and the right hand side is the output. The rules are applied nondeterministically, except that as noted in the side conditions, the rules (basic-1) and (basic-2) are used only if no other rules apply. Note that below, we will find it convenient to evaluate a sequence of MicroRogue expressions. After the first expression, each subsequent one is evaluated with respect to the rule list and hold expression resulting from evaluation of the previous expression. The rules do not mention the scoping operator α. To evaluate some expression using those rules, all subexpressions of the form $\alpha\, x. T$ are first rewritten to $[x'/x]T$, where x' is a fresh variable. No fresh variables are then introduced during evaluation.

5 Rogue in MicroRogue

This section explains how MicroRogue can be used to give a very concise implementation of Rogue. Noteworthy implementation techniques are used to handle the following two matters:

- **Interaction between congruence rules and reduction rules:** The left hand sides of reduction rules like $(r_1, r_2)@t \implies (r_1@t, r_2@t)$ overlap with congruence rules like one saying that an application is evaluated by evaluating its subexpressions in a certain way. We handle this overlap as follows in MicroRogue. The congruence rule is given higher priority than the reduction rule, simply by adding it to the global set of rules later. Once the congruence rule has rewritten subexpressions, it disables itself using `OFF`. To do this, it needs to know its unique identifier, which can be achieved simply by giving the rule a name when it is added. Then the congruence rule indicates that the hold expression, which is a possibly rewritten version of the application, should be further rewritten. This gives the reduction rule a chance to rewrite it. The reduction rule is then responsible for enabling the congruence rule again, using `ON`. If it could happen that no reduction rule applies, we can include a catch-all rule after the reduction rules, to turn the congruence rule on again. Alternatively, we can have the congruence rule turn itself back on after the recursive evaluation of the hold expression.
- **Explicit rule application:** Evaluation of an explicit rule application $(l \to r)@t$ is implemented in MicroRogue roughly as follows. We first dynamically add a catch-all rule which says that anything rewrites to `null`. Then we dynamically add the rule $l \to r$. We must add these rules in such a way that they will each be used at most once, and then both of them will be removed. This can be achieved, in a way described below. Finally, we

request that the target expression of the explicit application be rewritten. One of the two rules we added dynamically will apply. The rules will then both be removed, and rewriting will continue on the result (either `null` or $\sigma(r)$ for substitution σ with $\sigma(l) \equiv t$).

Figure 6 gives the definition of Rogue as a sequence of MicroRogue expressions, each terminated by ";;". Each MicroRogue expression is numbered for ease of reference. With the numbers stripped off, the code of Figure 6 is valid input to a prototype MicroRogue interpreter under development. One piece of syntactic sugar is being used, which is that `T_1 := T_2` stands for `T_1 -> T_2 !`, and `i : T_1 := T_2` for `i : T_1 -> T_2 !`. We let `:=` bind more loosely than all other operators except α. Note that the semantics of MicroRogue's `:=` operator are slightly different than Rogue's `:=` operator, since for MicroRogue's operator, the right hand side is not evaluated, while for Rogue's operator, the right hand side is evaluated. Except for expression 25, each numbered expression of Figure 6 causes a rule to be added to MicroRogue's global set of rules. Congruence rules are given names so they can be turned off and on. Reduction rules do not need names. We go through the definition in Figure 6 a piece at a time, starting with the simpler portions. For reasons of space, we omit an explanation of the rules for attributes (expressions 4, 5, 22, 23, and 24).

5.1 Arrow Expressions

Expressions 6 and 7 of Figure 6 give congruence rules for Rogue's two kinds of arrow expressions:

```
6.  x^y^ x->y := .0 ! ; DONE ;;
7.  x^y^ x=>y := .0 ! ; DONE ;;
```

These rules state that an arrow expression of either kind is to be rewritten to "`.0 ! ; DONE`". As codified in the (basic-1) and (basic-2) rules of Figure 5, when we attempt to find a matching rule for a target expression, the target becomes the new hold expression. Here, the rule (basic-2) will always be used, because an arrow expression E gets rewritten to "`.0 ! ; DONE`", which is distinct from E. So, (basic-2) says that we should recursively evaluate "`.0 ! ; DONE`" with the arrow expression as the new hold. Doing this causes first the (right-seq) rule to be used. The (subexpr) rule is used to rewrite (just) the left hand side of the arrow expression, and the (done) rule is then used to mark the hold expression as normalized. The rewritten hold expression is returned as the result of evaluating the original arrow expression.

5.2 Comma Expressions

Expressions 2, 10, 11, and 13 of Figure 6 are the ones relevant for evaluation of comma expressions. Note that bar expressions are handled similarly by expressions 3, 8, 9, and 14.

```
1.  x^y^ x@y  := ON APP1; ON APP2; DONE ;;
2.  x^y^ x,y  := ON COMMA1; DONE ;;
3.  x^y^ x|y  := ON BAR1; DONE ;;
4.  x^y^ x.y := null ;;
5.  DOT1 : x^y^ x.y := .0 ! ; .1 ! ; OFF DOT1 ; . ! ~; ON DOT1;;
6.  x^y^ x->y := .0 ! ; DONE ;;
7.  x^y^ x=>y := .0 ! ; DONE ;;
8.  x^ x|null := ON BAR1; x ;;
9.  x^ null|x := ON BAR1; x ;;
10. x^ x,null := ON COMMA1; x ;;
11. x^ null,x := ON COMMA1; x ;;
12. x^ null@x := ON APP1; ON APP2; null ;;
13. COMMA1 : x^y^ x,y  := .0 ! ; .1 !; OFF COMMA1; . ! ;;
14. BAR1 : x^y^ x|y  := .0 ! ; .1 !; OFF BAR1; . ! ;;
15. l^r^r2^t^ (l->r|r2) @ t :=
      PUSH; x^x -> (POP; ON APP1; ON APP2; r2 @ x) ! ;
              l -> (POP; ON APP1; ON APP2; r) ! ;
      t;;
16. r1^r2^t^ (r1,r2) @ t := ON APP1; ON APP2; r1 @ t, r2 @ t ;;
17. l^r^t^ (l->r) @ t := ON APP2; (l=>r) @ t ;;
18. APP2 : x^y^ x@y  := ON APP1; .1 ! ;
                        OFF APP1; OFF APP2 ; . ! ;;
19. l^r^r2^t^ (l=>r|r2) @ t :=
      PUSH; x^x -> (POP; ON APP1; r2 @ x) ! ;
              l -> (POP; ON APP1; r) ! ;
      t;;
20. l^r^t^x^ (l=>r)@t :=
      PUSH; x -> (POP; ON APP1; null) ! ;
            l -> (POP; ON APP1; r) ! ;
      t;;
21. APP1 : x^y^ x@y  := .0 ! ; OFF APP1 ; . ! ;;
22. x^y^set(x,y) := ON SETRULE1 ; ON SETRULE2; (x := y) ; y ;;
23. SETRULE2 : x^y^set(x,y) := .1.1 ! ; OFF SETRULE2; . ! ;;
24. SETRULE1 : x^y^z^set(x.y,z) := .1.0.0 ! ; .1.0.1 ! ;
                                   OFF SETRULE1; .! ;;
25. ENDCORE;;
```

Fig. 6. Definition of Rogue in MicroRogue

```
2.  x^y^ x,y  := ON COMMA1; DONE ;;
10. x^ x,null := ON COMMA1; x ;;
11. x^ null,x := ON COMMA1; x ;;
13. COMMA1 : x^y^ x,y  := .0 ! ; .1 !; OFF COMMA1; . ! ;;
```

Here we have an interaction between reduction rules (10 and 11) and a congruence rule (the part of 13 in parentheses). As mentioned above, our strategy is that the congruence rule will turn itself off after it has caused the subexpressions to be rewritten. The latter is achieved with "`.0 ! ; .1 !`", which first causes the left subexpression of the comma expression, and then the right, to be recursively evaluted. After the subexpressions are rewritten, the congruence rule turns itself off with the code `OFF COMMA1`, where `COMMA1` is just the name of the congruence rule. After the congruence rule has disabled itself, it requests that the hold expression be recursively evaluated ("`. !`"). At this point, the hold expression holds the comma expression with its subexpressions recursively evaluated. This comma expression will then be evaluated by using one of the rules (11), (10), or (2), tried in that order. Each rule first turns on the congruence rule and then returns an expression. Expression 2 is just a catch-all rule used to make sure the congruence rule gets turned back on.

5.3 Applications

The most complex functionally related subset of rules in Figure 6 is that for handling applications. Expressions 1, 12, 15, 16, 17, 18, 19, 20, and 21 are all concerned with applications. We will focus here just on handling applications of arrow expressions. Note first, though, that just as we earlier desired, applications of comma expression are handled by a simple rule (expression 16), with the modest addition of code to turn on some temporarily disabled rules:

```
16. r1^r2^t^ (r1,r2) @ t := ON APP1; ON APP2; r1 @ t, r2 @ t ;;
```

Let us consider the expressions for handling applications of arrow expressions:

```
1.  x^y^ x@y  := ON APP1; ON APP2; DONE ;;
17. l^r^t^ (l->r) @ t := ON APP2; (l=>r) @ t ;;
18. APP2 : x^y^ x@y  := ON APP1; .1 ! ;
                        OFF APP1; OFF APP2 ; . ! ;;
20. l^r^t^x^ (l=>r)@t :=
      PUSH; x -> (POP; ON APP1; null) ! ;
            l -> (POP; ON APP1; r) ! ;
      t;;
21. APP1 : x^y^ x@y  := .0 ! ; OFF APP1 ; . ! ;;
```

We use the technique described above for controlling the interaction between congruence rules and reduction rules. The situation is complicated, however, by the fact that applications `x@y` must be evaluated in stages. First we must see if `x` evaluates to a lazy arrow (`=>`) expression (or a bar expression beginning with a lazy arrow – but we here omit further consideration of that case, implemented by expression 19 of Figure 6, for simplicity). If `x` evaluates to a lazy arrow expression, we do not evaluate `y` but immediately use the appropriate reduction rule. If `x` evaluates to an eager arrow expression (`->`), then we must evaluate `y` before applying the appropriate reduction rule.

We address this additional complexity by using two congruence rules for applications `x@y`. The first is expression 21, which specifies that `x` should be recursively evaluated ("`.0 !`"), then that congruence rule should be disabled(`OFF APP1`), and finally evaluation should continue on the updated version of `x@y` ("`. !`"). If `x` has evaluated to a lazy arrow expression, then the rule added by expression 20 will match. We consider how this rule works below. Let us consider first, though, how evaluation proceeds if `x` has not evaluated to a lazy arrow expression. The next rule that can match (again, omitting consideration of bar expressions beginning with lazy arrow expressions) is the second congruence rule for applications, which is the part of expression 18 in parentheses. At this point in processing the original application `x@y`, we need to evaluate `y`. The congruence rule specifies this with "`.1 !`", but note a subtlety: we must re-enable the first congruence rule at this point, since recursive evaluation of a newly considered expression should start off with all rules enabled. So the congruence rule of expression 18 first turns the rule `APP1` back on. After the subexpression `y` has been recursively evaluated, both congruence rules (`APP1` and `APP2`) are disabled, and evaluation continues ("`. !`") on the application.

At this point, if no reduction rule applies, the catch-all rule of expression 1 will be used, and the expression will be marked as normalized. If, on the other hand, the application is now of the form `(l -> r) @ t`, the rule of expression 17 will be applied. This rule just returns `(l => r) @ t`, since applications of lazy and eager arrows are evaluated the same way if the target expression has already been evaluated; and turns the congruence rule `APP2` back on. We do the latter to maintain the invariant that when a rule is applied which occurs later in the list of rules than one of those congruence rules (`APP1` and `APP2`), all earlier congruence rules are enabled.

We come finally, then, to the rule of expression 20, for evaluating an application of the form `(l => r) @ t`. We use here the second technique mentioned at the beginning of Section 5 for dealing with explicit rule application. Consider again expression 20:

```
20. l^r^t^x^ (l=>r)@t :=
      PUSH; x -> (POP; ON APP1; null) ! ;
            l -> (POP; ON APP1; r) ! ;
      t;;
```

This says that evaluation of the application is to proceed as follows. We first push a new scope for dynamically added rules, and then add two rules. The first rule matches anything, since its left hand side `x` is just a variable. The second matches `l`, the left hand side of the rule we are trying to apply explicitly to target `t`. Finally, we request that `t` be recursively evaluated. If `t` matches `l`, then this causes the code `(POP; ON APP1; r)` to be executed. We pop the scope which contains (just) the two dynamic rules we added here. We then re-enable the congruence rule `APP1`, and then return σ(`r`), where σ is the matching substitution. If there is no such matching substitution for `l` and `t`, then the first rule (with left hand side `x`) we dynamically added applies. We again pop the scope for the two rules, re-enable `APP1`, and then return `null`.

5.4 Implementation

The current prototype interpreter for MicroRogue is able to evaluate Rogue programs with reasonable efficiency. For example, in a small number (62) of lines of Rogue code, a program which transforms lexical specifications using regular expressions into a nondeterministic finite automaton can be written. The MicroRogue interpreter, using the definition from Figure 6 of Rogue's operational semantics, can execute this program. The lexical specification for Java 1.4 can be transformed into an automaton with 423 states in 16 seconds using slightly under 4M of memory on a 1.2GHz Pentium 3 (Mobile) with 512K cache. The running time is a bit slow, but the memory usage is quite modest.

To achieve the current level of performance, the MicroRogue interpreter uses several advanced implementation techniques. These complicate the implementation, which nevertheless is currently around a modest 2000 (non-comment, non-blank) lines of C++. First, the global set of rewrite rules is indexed, currently just using a simple path indexing scheme (as described in, e.g., [11, Section 5]). Second, it is quite natural to find rules like several of the ones in Figure 6 whose right hand sides end in "`. !`", indicating that rewriting should continue on the current hold expression. The MicroRogue interpreter uses tail recursion when evaluating these expressions, to avoid allocating a stack frame. At present, the tail recursion must be explicitly indicated in the MicroRogue code by writing "`CONTINUE`" instead of "`. !`". This optimization allows expressions with lengthy evaluations to be evaluated without running out of stack memory.

Expressions are reference counted, but we do not insist that identical expressions are mapped to the same internal representation. This saves a lookup in a hash table for every creation of an expression. To keep memory usage down, expressions are implemented internally as C++ classes without any virtual methods. This saves 4 bytes that would otherwise be needed to point to the vtable structure. Compound expressions require three 4-byte words each: one for the left child, one for the right child, and one to hold the reference count and some flags. One flag is used to mark terms as closed which contain no variables; they do not need to be traversed during substitution. Substitution is carried out eagerly, since preliminary experiments with lazy substitution showed a slight decrease in performance.

A partial compiler for MicroRogue has also been implemented (in Rogue). MicroRogue rules are simply compiled into the C++ code that would otherwise be executed by the interpreter for them. This compilation is currently limited to the top-level of rules. Rules that are dynamically added by other rules are not compiled. The resulting partially compiled version of the MicroRogue program is then statically linked into the MicroRogue interpreter.

Applied to the definition of Rogue in MicroRogue, this partial compilation has cut around 20% of the running time from the interpretation of representative Rogue programs.

6 Conclusion and Future Work

We have seen how Rogue, an imperative version of the foundational Rewriting Calculus, can be defined in MicroRogue. MicroRogue's central ideas are to maintain a global set of rules which can be dynamically added, disabled or enabled, and pushed and popped in scopes; and to allow the order of evaluation to be explicitly defined by specifying positions of a hold expression to be recursively rewritten in place.

MicroRogue is a success as a simple rewriting language suitable for implementing languages like the Rewriting Calculus. The implementation of Rogue in MicroRogue together with the C++ implementation of MicroRogue is certainly not substantially more complicated than a Rogue interpreter implemented directly in C++, and is probably slightly less complicated. And having Rogue implemented in MicroRogue makes it much easier to experiment with the operational semantics of Rogue than if Rogue were directly implemented.

Nevertheless, MicroRogue is not yet satisfactory as a foundational system. To be foundational, we might like to establish meta-theoretic properties like: independence of the various features of a system; a connection to some other kind of fundamental system, like a logic of some kind; and a well-motivated denotational semantics. Thus, further work would be required to develop MicroRogue or something similar in spirit into a truly foundational rewriting language. Increasing performance remains another important aspect of future work.

Acknowledgments: Thanks go to the anonymous reviewers for very helpful suggestions on a previous draft of the 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] A. Stump and C. Schürmann. Logical Semantics for the Rewriting Calculus. In M. Bonacina and T. Boy de la Tour, editor, *5th International Workshop on Strategies in Automated Deduction*, 2004.

[2] A. Appel. Foundational Proof-Carrying Code. In *16th Annual IEEE Symposium on Logic in Computer Science*, 2001.

[3] A. Appel, N. Michael, A. Stump, and R. Virga. A Trustworthy Proof Checker. *Journal of Automated Reasoning, special issue on Proof-Carrying Code*, 31(3-4),

2003.

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

[5] C. Barrett, D. Dill, and A. Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In *14th International Conference on Computer-Aided Verification*, 2002.

[6] 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).

[7] 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).

[8] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1992.

[9] N. Dershowitz and D. Plaisted. *Rewriting*, chapter 9. Volume 1 of Robinson and Voronkov [10], 2001.

[10] A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier and MIT Press, 2001.

[11] R. Sekar, I. Ramakrishnan, and A. Voronkov. *Term Indexing*, chapter 26. Volume 2 of Robinson and Voronkov [10], 2001.

[12] A. Stump, A. Deivanayagam, S. Kathol, D. Lingelbach, and D. Schobel. Rogue Decision Procedures. In C. Tinelli and S. Ranise, editors, *1st International Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, 2003.

[13] E. Visser. Scoped dynamic rewrite rules. In Mark van den Brand and Rakesh Verma, editors, *Rule Based Programming (RULE'01)*, volume 59/4 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.