Treep: an explicitly multi-phased programming language

Eric Griffis

March 18, 2015

1 Introduction

Rule-based rewriting is ubiquitous in computing [DJ89, KDV90]. Many languages provide rewriting mechanisms for expressing user-defined code analyses and optimizations [JTH01, RO10, ERKO11], and many others adopt it as a model of computation [BvEVLP87, Dau92, Mes92, Sew98, Gra09].

We believe that rewrite rules can be a more convenient abstraction for building interpreters than functions or objects. Prior systems artificially restrict the rewriting model by forcing programs to either simulate functions with rewrite rules or vice versa. However, rewrite rules are not functions and so the abstraction inevitably leaks.

We introduce Treep, a language for building interpreters with constructs that complement the strengths of rewrite rules. Treep aims to simplify the creation, interpretation, and serialization of abstract syntax trees. The Treep design has three main goals.

1. Custom parsing and printing should be easier to do with native constructs than with external parser generators.

2. Rewrite rules should be as easy to express, apply, and compose as functions are in a functional language.

3. Extensibility is a top priority—primitive constructs should "stay out of the way" of user-defined constructs.

Section 2 introduces Treep syntax and semantics through a series of examples of progressively increasing complexity. Section 3 explains details of the prototype implementation relevant to the discussion in Section 1, including the command line arguments of the interpreter, minutia of its parser, and special forms available to Treep programs at run time. Section 4 formalizes Treep using operational semantics in a small-step style. Section 5 proposes future work and concludes.

```
term ::= term --> term | term | term
       | term |- term | term ||- term
       | term = term | ! term | term && term | term || term
       | #< term ># | #[ term ]#
       | atom
atom ::= _ | symbol | _ symbol | " char* " | #{ char* }#
       | #f | ## symbol
```

Figure 1: The abstract syntax of terms in Treep.

2 Treep by example

In this section, we introduce the Treep language and explore its benefits to programmers through a number of examples.

Treep offers a permissive lexical syntax and a list-based phrase syntax. Figure 1 gives the abstract syntax of terms. Section 3 explains the concrete syntax for symbols, which is similar to Scheme.

Treep models a computation as a series of rule-based rewrites. Throughout the evaluation process, some set of rules is actively transforming the program state. The active rule set may be empty and may change over time. A *rule* is a pair of terms joined by an arrow. We call the left term the *pattern* and the right term the *target*.

```
pattern --> target
```

When the rule above is active, it replaces all occurrences of `pattern` with `target`. A pattern is a template for matching other terms and may include variables and wildcards. A variable is a symbol prefixed by an underscore. For example, the following rules implement the familiar pair deconstructors of Haskell.

```
fst (_a,_) --> a                    snd (_,_b) --> b
```

Treep allows PCRE-compatible regular expressions in rule patterns. A *regexp* is a sequence of characters surrounded by special braces, `#{` and `}#`. A pattern with a regexp will not match unless the regexp coincides with a matching string. The rule set below implements a simple recognizer for the integers.

```
(recognize #{^-?[0-9]+$}# --> accept)
(recognize _              --> reject)
```

The first rule translates the list `recognize "-27"` into the symbol `accept`. Rule precedence

```
(#{^(.+) \Q+\E (.+)$}# --> \1 + \2)
(#{^(.+) \Q*\E (.+)$}# --> \1 * \2)
(#{^[0-9]+$}# --> #<\0>#)

|-

"1 + 2 * 3 + 4 * 5"
```

Figure 2: A regular arithmetic expression parser.

is set by the order in which the rules are defined, so the wildcard does not match unless the regular expression fails. Regexps may contain unnamed capture groups. A successful capture binds its result to a *backreference*, denoted by a number prefixed with a backslash. As in Perl, backreference numbering begins at 1 and increases in a breadth-first manner. Additionally, backreference `\0` refers to the entire regexp match.

Patterns may be guarded against indiscriminate matching. A *guard term* is a pair of terms joined by a vertical bar. When a rule pattern is a guard term, we call the left part a *guarded pattern* and the right part its *guard*. This form of pattern will match a term only if its guarded pattern matches the term and its guard does not evaluate to `#f`, the special symbol representing Boolean false. In the following rules, the pattern on the left matches anything while the pattern on the right matches nothing.

```
_ --> 17                    _ | #f --> 17
```

Treep is an explicitly multi-phased language, which means programs must specify the boundary between each phase of evaluation. A *phase* is a pair of terms joined by a turnstyle. A phase is *proper* if it is the only top-level term in the program. We call the left part of a proper phase the *environment* and the right part the *stage*. A rule set is active iff it is the environment of a running program. When the stage of a proper phase reaches a normal form with respect to its environment,[1] the phase reduces to its stage—the environment and turnstyle disappear.

Figure 2 demonstrates a simple two-phase program. Phase one defines a parser for regular arithmetic expressions. The first two rules establish the usual order of operations for addition and multiplication. The third rule uses a parse term to transform a sequence of digits into a number. A *parse term* is a list surrounded by special angle brackets, `#<` and `>#`. When the list inside a parse term evaluates to a string, the built-in parser consumes it and replaces the parse term with the result. Phase two holds the input string, `"1 + 2 * 3 + 4 * 5"`. Figure 16 gives the complete derivation of this program.

The phase operator is right-associative and has very low precedence. Occasionally, we would like to force evaluation of a term early or in an unconventional order. The *force* operator is a turnstyle with a double vertical bar (`||-`). It denotes a phase boundary, but short circuits the default order of evaluation. We call a forced phase a *sub-evaluation*.

[1]In Treep, the concept of normal form only makes sense within the context of an environment.

```
(_a + _b --> ##+ a b)
(_a * _b --> ##* a b)

|-

calc0 --> ##load "calc0.trp"

|-

(calc0 ||- "1 + 2 * 3") + (calc0 ||- "4 * 5")
```

Figure 3: A regular arithmetic expression evaluator.

Figure 3 demonstrates the use of force to simulate object-oriented message passing. Suppose file `"calc0.trp"` contains the first phase of Figure 2. Phase one specifies how to evaluate infix arithmetic operations. Phase two imports the parser. The load operation has very high precedence, so the parser code is subject to rewriting by the rules in phase one. These rules transform the parser into an evaluator. Phase three sub-evaluates two arithmetic expressions and combines the results. The final result is `27`.

The program in Figure 3 uses pragmas to perform several basic operations. A *pragma* is a special symbol prefixed with two sharps. Pragmas enable communication with the host platform. Treep relies on pragmas for most non-rewrite operations, including addition (`##+`) and multiplication (`##*`). See section 3 for details on pragmas.

The remaining four Treep constructs enable the creation of simple key-value stores called maps. A *map* is a dictionary that binds symbols to terms. When a map comprises the environment of a proper phase, it provides a simple form of explicit substitution. This is the same mechanism used internally to substitute variables in rule targets after successful pattern matches. A *map constructor* is a pair of terms joined by an equals sign. The left part is a pattern. We call the right part the *source*.

```
pattern = source
```

When `source` reaches a normal form, it is compared against `pattern`. Upon success, the map constructor reduces to a map containing the captured variables, which may be empty. Upon failure, it reduces to `#f`.

The final three constructs give Boolean-like semantics to maps. Intuitively, the special symbol `#f` is "false" and any other normal form is "true." The *negation* (`!`) prefix replaces `#f` with the empty map and anything else with `#f`. For example, the two left-most terms below evaluate to `#f`, while the two on the right evaluate to the empty map.

```
x = y          !(x = x)          !(x = y)          x = x
```

A *disjunction* is a pair of terms joined by two vertical bars (`||`). Evaluation of disjunctions is short-circuiting in the following sense. The left part evaluates first; if it evaluates to `#f`, the

```
(_a * _b --> ##* a b)
(_a - _b --> ##- a b)
(_a / _b --> ##div a b)
(_a % _b --> ##mod a b)

|-

(_b ^ 0 --> 1)
(_b ^ 2 --> b * b)
(_b ^ _p | 0 = p % 2 --> (b ^ (p / 2)) ^ 2)
(_b ^ _p --> b * (b ^ (p - 1)))

|-

2 ^ 7
```

Figure 4: An integer power function.

disjunction reduces to the right part—the left part disappears. If the left part evaluates to any other normal form, the disjunction reduces to it—the right part disappears, unevaluated. Below, the left term evaluates to the empty map and the right term evaluates to `#f`.

```
x = y || x = x                    x = y || y = x
```

A *conjunction* is a pair of terms joined by two ampersands (`&&`). Evaluation of conjunctions is also short-circuiting, but in a slightly more complex way. If the left part evaluates to `#f`, the conjunction reduce to `#f`—the right part disappears, unevaluated. If the left part evaluates to any other normal form that is not a map, the conjunction reduces to the right part—the left part disappears. If the left part evaluates to a map, then the right part evaluates before the left part disappears. If the right part also evaluates to a map, the conjunction reduces to a map containing the union of the two. Otherwise, the conjunction reduces to the right part, as if the left part were any other non-false normal form. For example, the terms below evaluate to `#f`, `22`, and the empty map, respectively.

```
#f && x = x          x = x && 22          x = x && y = y
```

Negations, conjunctions, and disjunctions enable incremental and conditional construction of maps. They are designed to enrich the expressive power of pattern guards. For example, the program in Figure 4 implements the well-known "exponentiation by squaring" optimization with the map constructor in the guard of its third rule. The final result of this program is `128`, and Figure 17 gives the corresponding derivation.

As a final note, Figure 5 demonstrates another use of force; this time, as a code import mechanism. Suppose file `"pow.trp"` contains the first two phases of Figure 4. Then stage one of Figure 5 loads the file, evaluates its contents, and sets the resulting rule set as the

```
(||- ##load "pow.trp")

|-

_a fifth --> a ^ 5
```

Figure 5: A specialized power function.

environment. The final result of this program is a rule for which a recursive application of the power operator has been unrolled. Figure 18 shows the derivation of the rule. This example occurs in the Multi-Stage Programming (MSP) literature as an illustration of user-defined program specialization [Tah99, RO10], and the ease with which it is expressed here suggests a connection between Treep and MSP.

2.1 An HTTP request parser

Behavior of the Treep parser can not be altered directly. Instead, parser "extensions" are just Treep programs that use regular expressions to decompose strings into smaller fragments that can be altered and then consumed by the built-in parser. Perl is another language designed for processing strings with regular expressions, but Perl and Treep have dramatically different designs.

Figure 6 illustrates a regular expression-based implementation of an HTTP request parser in Perl. The `parse` procedure performs a number of regular expression matches, each extracting a different part of the serialized request, and returns a reference to a hash that names those parts when applied to a valid request. This approach fully exploits the convenience of Perl string processing constructs and the implicit variable to produce compact and reasonably efficient code. It can handle any number of headers and tolerates empty request bodies.

However, this approach has a number of disadvantages. First, the procedural style is verbose. For example, the parser in Figure 6 needs two short-lived temporary variables, `$h` and `$b`, and enforcing tight limits on the scope of these temporaries would require nested blocks or extra procedure calls. Second, heavy reliance on automatic run-time optimization makes reasoning about performance difficult. For example, the effects of tight variable scoping on memory conservation are not always clear because the default memory manager may not free storage at end of scope [X15]. Finally, Perl prioritizes string processing over user-defined (e.g., algebraic) data types, which complicates other aspects of interpreter construction. For example, although serializing a hash takes a simple string interpolation, matching against a composite data structure involves nested `if` statements or dispatch tables.

Figure 7 shows how an equivalent parser can be implemented in Treep. The program is divided into three distinct phases. The first phase defines a rule that binds the symbol `Next` to a regexp that matches a line terminator sequence and captures the remainder of the request in a backreference. Phase two defines regexps for deconstructing the initial request

```
sub parse {
    my ($h, $b) = split /\r\n\r\n/s, $_[0], 2;
    my @head = split /\r\n/, $h;

    my %req;
    if ((shift @head) =~ m{^([^ ]+) ([^ ]+) HTTP/1\.1$}) {
        %req = (method => $1, url => $2, body => $b);
    } else { die "bad request" }

    foreach (@head) {
        if (/^([^:]+): (.+)/) {
            $req{$1} = $2;
        } else { die "bad header" }
    }

    return \%req;
}
```

Figure 6: An HTTP request parser implemented in Perl.

and subsequent header lines. A list of regexps concatenates into a single larger regexp, so each of the regexps bound in phase two will contain three capture groups by the time phase one ends. Phase three specifies the complete parser logic in a functional style.

The Treep implementation of the parser resolves the problems of the Perl implementation. The declarative style permits a concise, high-level description of the program logic separate from the details of the tokenizer, which eliminates the need for temporary variables and explicit input stack management. Additionally, Treep's rule-based syntax and semantics naturally maintain tight scoping of pattern variables without extra code.

Further, the Treep interpreter does not perform opaque optimizations at run time. Treep does not manage a significant amount of memory implicitly—the loaded program *is* the heap—so the memory consumed by a program is evident in its code. Moreover, all of the programmer-defined optimizations—in this case, the rules of phases one and two—are defined in native Treep code. Finally, Treep provides a consistent syntax for deconstructing all primitive terms, including strings.

Regular languages like the HTTP request format are ubiquitous in modern software systems because they are convenient for driving machine-machine interactions. Further examples include SMTP headers, web server logs, server configuration files, and user-facing input scrubbers and validation engines. Regular languages may also be combined to form more complex languages; for instance, by embedding URL-encoded form data in an HTTP POST request body. Composing regular language parsers in Treep is convenient and intuitive. However, sometimes we need more expressive power than regular languages offer, particularly when dealing with human-machine interactions. The remaining examples address this

```
Next --> #{\r\n((?:.|\r|\n)*)$}#

|-

(ReqLine --> #{^([^ ]+) ([^ ]+) HTTP/1\.1}# Next)
(Header  --> #{^([^:]+): (.+)}# Next)

|-

(parse ReqLine --> head (_method = \1 && _uri = \2) \3)
(parse _       --> reject "bad request")

(head _req Header --> head (req && #<"_" \1># = \2) \3)
(head _req Next   --> req && _body = \1)
(head _    _      --> reject "bad header")
```

Figure 7: The HTTP request parser implemented in Treep.

concern.

2.2 Parenthesized expressions

The calculator from Figure 3 is of limited use because the order of operations is fixed by the interpreter. A common strategy for overcoming this limitation is to introduce balanced parentheses to denote sub-expression boundaries. Deciding whether parentheses are balanced is simply a matter of keeping count, but regular expressions will not do this for us.

Figure 8 gives a recognizer for the language of balanced parentheses in Treep. Phase one defines the interface to the host platform along with a rule for defining literal token consumers. Phase two defines the tokenizer, and phase three implements the core logic. The `loop` rules examine the input and adjust the depth variable `d`. If the final value of `d` is `0`, the input is accepted. For any other final value of `d`, or if `d` ever becomes negative, the input is rejected. The `recognize` rule initializes the loop. Figure 19 shows the derivation of this program.

Figure 9 shows how an equivalent parser can be implemented in Haskell. The functions are similar to their Treep counterparts, except that the `loop` function decomposes the input as a character list instead of by tokenizing with regular expressions. In this small example, the Haskell program is more compact and probably more efficient than the Treep version because it does not use regular expressions. We also benefit from the presence of the static type checker.

The balanced parentheses recognizers are simple because the amount of state—a single counter—is minimal. Adding parenthesized sub-expressions to our arithmetic expression parser will require significantly more complex state. Below is a common formulation of the

```
(_a ++ --> ##+ a 1)
(_a -- --> ##- a 1)
(_a > _b --> ##> a b)
(Literal _c --> #{^ *}# c #{ *(.*)$}#)

|-

(LP --> Literal "(")
(RP --> Literal ")")

|-

(recognize _input --> loop input 0)

(loop LP _d          --> loop \1 (d ++))
(loop RP _d | d > 0 --> loop \1 (d --))
(loop ""  0          --> accept)
(loop _   _          --> reject)

|-

recognize "() (())"
```

Figure 8: A balanced parentheses recognizer implemented in Treep.

```
recognize :: String -> Bool
recognize input = loop input 0

loop :: String -> Int -> Bool
loop "" d = d == 0
loop (c:cs) d = case c of
  ' '         -> loop cs d
  '('         -> loop cs $ d + 1
  ')' | d > 0 -> loop cs $ d - 1
  _           -> False

main = putStrLn $ show $ recognize "() (())"
```

Figure 9: The balanced parentheses recognizer implemented in Haskell.

desired language as an unambiguous context-free grammar (CFG) with left recursion, which we use as a reference for building a simple shift-reduce parser.

$$\begin{aligned} exp &::= exp \;\texttt{+}\; fac \;\Big|\; fac \\ fac &::= fac \;\texttt{*}\; trm \;\Big|\; trm \\ trm &::= num \;\Big|\; \texttt{(}\; exp \;\texttt{)} \end{aligned}$$

We can convert this CFG systematically into an equivalent two-state pushdown automata (PDA) by, e.g., Sipser [Sip12]. One state corresponds to an active main loop and the other to acceptance of the input. The parser state is the PDA stack. Each transition from the loop state to itself either shifts an element onto the stack or reduces some elements at the top of the stack.

Figure 10 shows the PDA-based parser implemented as a Treep program. The rule in phase one is copied directly from the recognizer, as are the first two rules of phase two. The next three rules define token consumers for the operators and numbers. The final three rules of phase two define a small DSL for specifying our parser logic. The first rule defines a three-argument stack-shifting abstraction named `shift`. The first argument is a token consumer, and the second argument specifies which backreference contains the rest of the input when the token consumer succeeds. The third argument defines the term to shift. The final two rules of phase two define an overloaded "reduce" operator (`::=`) that matches all arguments but the last against the elements at the top of the stack and, on success, replaces them with the term defined by the last argument. Figure 20 gives the derivation of this program.

This program is interesting in a number of ways. In three rules, we were able to implement a DSL for building basic shift-reduce parsers. The program is substantially more readable than without it, and the custom syntax emphasizes the relationship between the original CFG and the final code. The order of the "reduce" rules is precisely the reverse of the CFG productions. This custom syntax is actually a by-product of a series of standard agile prototype-refactor cycles, obtained by implementing parts of the main loop directly and capturing the patterns in the code as they emerged.

However, the implementation has at least one major drawback. Among the `shift` rules, order is completely irrelevant for `Num` and `LP` but supremely relevant for the others. The reason is evident in the CFG, but only after careful consideration. Some of the productions "reduce" by re-tagging the top of the stack with an equivalent but lower-precedence production as a catch-all. For instance, the rule above `shift Plus` re-tags a `fac` as an `exp` if the preceding rule fails to "reduce" the top of the stack to an addition operation. If the `shift Plus` rule is moved one line up, the parser will always `shift` the `+` before the left part can be tagged as an `exp` and thus will fail on any input with an addition. This kind of subtlety complicates the implementation of more substantial parsers. We leave this issue open for future work.

```
Literal _c --> #{^ *}# c #{ *(.*)$}#

|-

(LP    --> Literal "(")
(RP    --> Literal ")")
(Plus  --> Literal "+")
(Times --> Literal "*")
(Num   --> #{^ *([0-9]+)(.*)$}#)

(shift _rx _i _R --> loop rx _s             --> loop i (R s))
(_L        ::= _R --> loop _i (L _s)         --> loop i (R s))
(_N _M _L ::= _R --> loop _i (L (M (N _s))) --> loop i (R s))

|-

(parse _input --> loop input $)

((num _n)      ::= (trm n))
([ (exp _e) ] ::= (trm e))
(shift Num \2 (num #<\1>#))

((fac _f) * (trm _t) ::= (fac (f * t)))
((trm _t)            ::= (fac t))
(shift Times \1 *)

((exp _e) + (fac _f) ::= (exp (e + f)))
((fac _f)            ::= (exp f))
(shift Plus \1 +)

(shift LP \1 [)
(shift RP \1 ])

(loop "" ((exp _e) $) --> accept e)
(loop _  _            --> reject)

|-

parse "1 * (2 + 3)"
```

Figure 10: A parenthesized arithmetic expression parser implemented in Treep.

2.3 A Forth interpreter

Then final example in this section demonstrates a rudimentary Forth interpreter implemented in Treep. Forth is a stack-based, extensible, Turing complete programming language with an extremely simple syntax [Pel11].

Figure 11 shows a Forth interpreter implemented as a Treep program. Phases one and two define the lexer in the style of previous examples. Phase three defines the Forth evaluator as a multi-stepper, where each step consumes a single element from the head of the input. The `step` rules implement five kinds of statements and a stack dumper for debugging invalid inputs. Evaluation ends when the input is exhausted. The final phase runs the evaluator on a Forth program that defines a function `add-two` and then applies it to the number `9`. The program prints to the console: `9 + 2 = 11`.

Each `step` rule looks for a specific form at the head of the input. In Forth jargon, the first two rule handle words defined in the dictionary. The `b` rule defines four built-in words: addition, multiplication, duplication, and a destructive print statement. This implementation keeps built-in words separate from user-defined words, but handles user-defined words first so that built-ins can be redefined by user code. The third `step` rule prints a string to the console. The fourth `step` rule pushes a number onto the stack, the fifth handles new word definitions, and the sixth detects the end of input.

The first `dump` rule uses a print term to serialize an element of the stack. A *print term* is a list surrounded by special square brackets, `#[` and `]#`. When the list inside a parse term reaches a normal form, the built-in printer consumes it and replaces the print term with the result.

3 Implementation

Treep is implemented in Haskell 2010 and compiled with GHC version 7.8.4. The built-in parser is generated by Alex and Happy. Additional dependencies included libraries for regular expressions and some monad transformers.

The prototype is provided as a Cabal source package that may be built and installed by running `cabal install` from the root directory of the unpacked tarball. This will build and install an executable file named `treep`. Executing this command with no command-line arguments starts an interactive REPL. To enable evaluation traces at the command line, add the `-t` switch. To evaluate a file from the command line, specify the path to the file as the final argument. The source package contains several programs, including all of the programs in this document, in the `examples` directory. All errors are fatal, including run-time parse failures and failed pragma calls.

The parser recognizes four kinds of symbol.

- A *number* is a sequence of digits, prefixed by a hyphen (`-`) if negative. Leading zeros are ignored.
- A *word* is a sequence of alphanumeric characters. Words may contain modern Greek

```
Next --> #{(?: (.*))?$}#

|-

(Wrd --> #{^([^ ]+)}# Next)
(Num  --> #{^([0-9]+)}# Next)
(Str  --> #{^\.’ ([^’]*)’}# Next)
(Def  --> #{^: ([^ ]+) ([^;]+ );}# Next)

|-

(b --> ("+"   --> $ ($ (_A (_B _s)) --> (##+ A B) s))
       ("*"   --> $ ($ (_A (_B _s)) --> (##* A B) s))
       ("dup" --> $ ($ (_A _s) --> A (A s)))
       ("."   --> $ ($ (_A _s) | ##print (" " #[A]#) --> s)))

(step Wrd _s _d | $ _f = (d ||- \1) --> step (f \2) s d)
(step Wrd _s _d | $ _f = (b ||- \1) --> step \2 (f ||- $ s) d)
(step Str _s _d | ##print (" " \1)  --> step \2 s d)
(step Num _s _d --> step \2 (#<\1># s) d)
(step Def _s _d --> step \3 s (d (\1 --> $ \2)))
(step ""  _  _  --> )

((puts _A --> ##print (A "\n"))
 (mark _A --> ">>>" A "<<<")
 ||-
 (step  Wrd _s _ --> puts ((dump s) " " (mark \1) " " \2))
 (step _src _s _ --> puts ((dump s) " " src (mark ""))))

(dump (_A _s) --> (dump s) " " #[A]#)
(dump ()      --> "")

(eval _src --> step src () ())

|-

eval ": add-two dup . .’ + 2 =’ 2 + . ; 9 add-two"
```

Figure 11: A forth interpreter implemented in Treep.

`##+`	`##*`	`##-`	`##div`
`##mod`	`##>`	`##<`	`##>=`
`##<=`	`##file`	`##load`	`##print`

Table 1: Pragmas supported by the prototype interpreter.

letters (A-$\Omega\alpha$-ω) and underscores (`_`), may have any number of trailing apostrophes (`'`), and must not start with a digit or an underscore.

- An *operator* is a sequence of *punctuation* characters, or non-word characters in the ASCII hex code range 20-7E except for the sharp sign (`#`).
- A *special* is like any other symbol, but either starts or ends with a sharp sign.

Symbol classification is purely a pragmatic matter. However, the prototype recognizes integers as a special kind of symbol to simplify interaction with the Haskell run time.

To facilitate program debugging, the built-in printer serializes maps as if they were simple rule sets. Maps are printed with surrounding special braces annotated with an M to make them visually distinct from actual rule sets.

```
#M{(x --> 5) (y --> 4)}#
```

The built-in parser does *not* recognize this notation.

Table 1 shows the pragmas recognized by the prototype. The first nine correspond to the Haskell operator of the same name and expect a pair of integer arguments. The numeric operators return an integer and the inequality tests return either `#f` or the empty map. The `##file` pragma expects a single string argument containing a file path and returns the contents of the file as a string. The `##load` pragma is similar, but also parses the file. In other words, `##load "program.trp"` is equivalent to `#< ##file "program.trp" >#`. The `##print` pragma expects a single string argument, which it writes to the standard output, and returns the empty map.

4 Formalism

The behavior of Treep programs is determined by the formal semantics, specified here as structural operational semantics in a small-step style, over terms of the formal syntax in Figure 12. The operational semantics occasionally refer to one of the sets of all terms of a particular form.

$\mathcal{A}$ = atoms	$\mathcal{Z}$ = character sequences	Σ = rule sets
$\mathcal{X}$ = variables	$\mathcal{G}$ = guard terms	Γ = maps

The formal semantics make liberal use of the following convenient but slightly abusive notations. Let t be a term in the formal language. We call t a *normal form* if t makes no progress with respect to the formal semantics. We may write $t \rightsquigarrow \bot$ to indicate t is a normal

$$\begin{array}{rcll} a & ::= & x & \text{symbol} \\ & \mid & v^{(x)} & \text{variable} \\ & \mid & \diamond & \text{wildcard} \\ & \mid & z & \text{regexp/string} \\ & \mid & \varphi & \text{pragma} \\ & \mid & \text{false} & \text{boolean} \\ & \mid & \sigma & \text{rule set} \\ & \mid & \gamma & \text{map} \end{array} \qquad \begin{array}{rcll} t & ::= & a & \text{atom} \\ & \mid & [t]_I & \text{parse} \\ & \mid & [t]_O & \text{print} \\ & \mid & t|t & \text{guard} \\ & \mid & t \rightarrow t & \text{rule} \\ & \mid & t \cdots t & \text{list} \\ & \mid & t \vdash t & \text{phase} \\ & \mid & t \Vdash t & \text{force} \\ & \mid & t \cong t & \text{match} \\ & \mid & \neg t & \text{negation} \\ & \mid & t \wedge t & \text{conjunction} \\ & \mid & t \vee t & \text{disjunction} \end{array}$$

Figure 12: The formal syntax of Treep.

form. Alternatively, we may write v to indicate a normal form whenever the following is true.

$$\exists\ t \text{ such that } v = t \text{ and } t \rightsquigarrow \bot$$

The operational semantics rely heavily on these notations to indicate terms in normal form.

4.1 Notation

The following notations assist in defining the formal semantics of the language. Denote by $\rightsquigarrow^*$ the reflexive-transitive closure of the small-step relation, and define $t \Downarrow t'$ the *sub-evaluation* of t to t' as follows.

$$t \Downarrow t' \iff \exists\ t'' \text{ such that } t \rightsquigarrow^* t'' \text{ and } t'' \rightsquigarrow \bot$$

Denote by $\phi(\varphi, v_1 \cdots v_n)$ a call to the host platform identified by pragma φ, with arguments $v_1, \ldots, v_n$. Denote by $t_1 \oplus t_2$ the concatenation of terms t_1 and t_2, defined as follows,

$$t_1 \oplus t_2 = \begin{cases} t_1 \cup t_2 & \{t_1, t_2\} \subset \Sigma \text{ or } \{t_1, t_2\} \subset \Gamma \\ \overline{t_1 t_2} & \{t_1, t_2\} \subset \mathcal{Z} \end{cases}$$

where $\overline{z_1 z_2}$ is the regexp formed by concatenating the bodies of regexps z_1 and z_2. Denote by $t_1 \cong t_2 = \gamma$ a match between pattern t_1 and term t_2 that produces the map γ. The match relation is defined as follows,

$$t_1 \cong t_2 = \begin{cases} \gamma & \{t_1, t_2\} \subset \mathcal{Z} \text{ and } \mu(z_1, z_2) = \gamma \\ \{x \mapsto t_2\} & t_1 = v^{(x)} \\ \varnothing & t_1 = \diamond \\ \varnothing & \{t_1, t_2\} \subset \mathcal{A} \text{ and } t_1 = t_2 \end{cases}$$

where $\mu(z_1, z_2) = \gamma$ if regexp z_1 matches string z_2 and captures bindings γ. Denote by $\gamma(\alpha; t)$ the substitution in term t of the symbols bound by γ that are not also elements of $\alpha \subset \mathcal{X}$, defined as follows,

$$\gamma(\alpha; t) = \begin{cases} t' & t = x \notin \alpha \text{ and } x \mapsto t' \in \gamma \\ \gamma(\alpha; t_1) \to \gamma(\alpha \cup \text{vars}(t_1); t_2) & t = t_1 \to t_2 \end{cases}$$

where $\text{vars}(t) = \{\text{variables in } t\}$. Denote by $\sigma(t)$ the rewriting by rule set σ of term t, defined by $\sigma(t) = t' \iff$ one of the conditions below is satisfied.

- $\exists\ t_1 \to t_2 \in \sigma$ such that $t_1 \notin \mathcal{G}$ and $t_1 \cong t = \gamma$ and $\gamma(t_2) = t'$
- $\exists\ t_1 | t_2 \to t_3 \in \sigma$ such that $t_1 \cong t = \gamma_1$ and $\gamma_1(t_2) = t_2'$ and $\sigma \models t_2' \Downarrow \gamma_2$ and $\gamma_1 \oplus \gamma_2 = \gamma_3$ and $\gamma_3(t_3) = t'$
- $\exists\ t_1 | t_2 \to t_3 \in \sigma$ such that $t_1 \cong t = \gamma_1$ and $\gamma_1(t_2) = t_2'$ and $\sigma \models t_2' \Downarrow t_2''$ and $t_2'' \notin \Gamma \cup \{\text{false}\}$ and $\gamma_1(t_3) = t'$

In the first case, variables captured by an unguarded rule pattern will be bound in the target. In the other two cases, any variables captured by a guard will also be bound in the target.

Finally, the semantics use three helper functions: $\text{parse}(z)$ applies the built-in parser to string z, $\text{print}(t)$ applies the built-in printer to term t, and $\text{compile}(t)$ verifies that t is a rule set.

4.2 Evaluation

The evaluation semantics are partitioned into two distinct parts: expansion and reduction. These names do not imply a deep or meaningful connection to the particulars of any other language. Intuitively, the separation of expansion and reduction is to protect rules, guards, and print terms from premature reduction.

Figure 13 details the core evaluation semantics. To summarize, evaluation of a given term proceeds as follows. First, try to expand the term. If expansion fails, try to reduce the term. If reduction fails *and* the term is a phase *and* its left part is a rule set, the current phase ends and evaluation of the term as the next phase begins. Otherwise, the current phase ends and the term is the final result.

4.2.1 Expansion

All built-in parsing, concatenation, pragma calls, sub-evaluations, and rewrites occur during expansion. Of these, parsing is always first and rewriting is always last. Expansion is defined recursively on the structure of all non-atomic terms. Consequently, rewriting occurs depth-first from left to right. Figure 14 details the non-recursive parts of the expansion semantics.

$$\frac{\sigma \models t \rightsquigarrow_E \sigma \models t'}{\sigma \models t \rightsquigarrow \sigma \models t'} \qquad \frac{\sigma \models t \rightsquigarrow_E \bot \qquad \sigma \models t \rightsquigarrow_R \sigma \models t'}{\sigma \models t \rightsquigarrow \sigma \models t'} \qquad \frac{\text{compile}(v_1) = \sigma_1}{\sigma \models v_1 \models v_2 \rightsquigarrow \sigma_1 \models v_2}$$

$$\frac{}{\sigma \models v \rightsquigarrow v}$$

Figure 13: The core evaluation semantics.

$$\frac{\text{parse}(z) = t}{\sigma \models [z]_I \rightsquigarrow_E \sigma \models t} \qquad \frac{t \notin \mathcal{Z} \qquad \sigma \models t \rightsquigarrow_E \sigma \models t'}{\sigma \models [t]_I \rightsquigarrow_E \sigma \models [t']_I}$$

$$\frac{v_1 \oplus v_2 = v'}{\sigma \models v_1\ v_2\ t_3 \cdots t_n \rightsquigarrow_E \sigma \models v'\ t_3 \cdots t_n} \qquad \frac{\phi(\varphi_1, v_2 \cdots v_n) = t'}{\sigma \models \varphi_1\ v_2 \cdots v_n \rightsquigarrow_E \sigma \models t'}$$

$$\frac{\text{compile}(t_1) = \sigma_1 \qquad \sigma_1 \models t_2 \Downarrow t'}{\sigma \models t_1 \mid\models t_2 \rightsquigarrow_E \sigma \models t'} \qquad \frac{\sigma(t) = t'}{\sigma \models t \rightsquigarrow_E \sigma \models t'}$$

Figure 14: The expansion semantics.

4.2.2 Reduction

All explicit substitutions, map constructions, Boolean-like operations, and built-in printing occur during reduction. Of these, printing is always last. Reduction is defined recursively on all non-atomic terms *except* for rules, forced phases, and the patterns of map constructor. Figure 15 details the non-recursive parts of the reduction semantics.

5 Future work and conclusions

Treep attempts to balance the needs of implementers and users of domain-specific languages. The examples of Section 2 show that, for regular languages, we have succeeded. However, the examples also demonstrate that the Treep design requires more attention before we can say the same for more expressive languages. Specifically, we would like to eliminate the subtleties that arise when converting context-free grammars into shift-reduce parsers. To this end, we plan to explore new ways to extend pattern matching. For example, Treep can not yet deconstruct lists of variable or unknown length—a basic feature of other list-oriented meta-languages such as Lisp or Scheme. Similarly, but less critically, we would like to tune the syntax of Treep constructs to reduce the need for parentheses, perhaps by introducing constructs like Haskell's dollar-sign and colon operators, or by allowing parser hints through indentation. For real-world applications, this level of sophisticated syntax may be worth

$$\frac{\gamma(\varnothing; t) = t'}{\sigma \vdash \gamma \vdash t \leadsto_R \sigma \vdash t'} \qquad \frac{v_1 \cong v_2 = \gamma}{\sigma \vdash v_1 \cong v_2 \leadsto_R \sigma \vdash \gamma} \qquad \frac{v_1 \not\cong v_2}{\sigma \vdash v_1 \cong v_2 \leadsto_R \sigma \vdash \text{false}}$$

$$\frac{}{\sigma \vdash \neg\text{false} \leadsto_R \sigma \vdash \varnothing} \qquad \frac{v \neq \text{false}}{\sigma \vdash \neg v \leadsto_R \sigma \vdash \text{false}} \qquad \frac{}{\sigma \vdash \text{false} \wedge t \leadsto_R \sigma \vdash \text{false}}$$

$$\frac{}{\sigma \vdash v \wedge \text{false} \leadsto_R \sigma \vdash \text{false}} \qquad \frac{v_1 \oplus v_2 = v'}{\sigma \vdash v_1 \wedge v_2 \leadsto_R \sigma \vdash v'} \qquad \frac{v_1 \neq \text{false} \qquad v_1 \not\oplus v_2}{\sigma \vdash v_1 \wedge v_2 \leadsto_R \sigma \vdash v_2}$$

$$\frac{}{\sigma \vdash \text{false} \vee t \leadsto_R \sigma \vdash t} \qquad \frac{v \neq \text{false}}{\sigma \vdash v \vee t \leadsto_R \sigma \vdash v} \qquad \frac{\text{print}(v) = z}{\sigma \vdash [v]_O \leadsto_R \sigma \vdash z}$$

Figure 15: The reduction semantics.

the effort. We are also interested in ways to extend the design to enhance modularity. For example, there is no way to prevent arbitrary code from using the Forth interpreter's `step` rules directly, which can lead to brittleness. We would also like to explore support for socket-based IO and regexp matching for potentially infinite character streams.

Because this experiment focuses on the pragmatic consequences of extreme syntactic flexibility on an extensible language design, we address neither theoretical aspects of the design nor performance of the implementation. An obvious question of theoretical interest is whether Treep programs can be proven sound statically, perhaps with a mechanism like the Knuth-Bendix convergence test [KB83]. Similarly, Treep's explicit phasing and rewriting constructs make for a unique coding experience that encourages the growth of concise and optimized code organically, and we expect a basic collection of program analysis tools could enable in-editor detection and exploitation of usage patterns as they emerge.

With respect to run-time performance, we have no benchmarks for the prototype. We can safely assume that Treep performance is at least an order of magnitude worse than a mature language like Perl or Haskell. The prototype rewriting engine naively iterates through all the rules in a rule set because the "compile" helper merely detects rule sets without doing any actual compiling! Instead, we would like to compile active rule sets into automata that avoid spurious re-checking of shared pattern prefixes. For example, we should be able to check if the first element of a list is the symbol `step` once and, if not, ignore all of the active `step` rules. We would also like to integrate regexps into these automata, and extend them with named capture groups.

In conclusion, Treep offers a powerful and convenient alternative to Perl for regular language processing, but falls short of Haskell when processing more complex languages. Still, the process of growing efficient code in Treep can lead to valuable insights into the design of domain-specific languages, independent of the language used to implement the design.

References

[BvEVLP87] TH Brus, Marko CJD van Eekelen, MO Van Leer, and Marinus J Plasmeijer. Clean—a language for functional graph rewriting. In *Functional Programming Languages and Computer Architecture*, pages 364–384. Springer, 1987.

[Dau92] Max Dauchet. Simulation of Turing machines by a regular rewrite rule. *Theoretical Computer Science*, 103(2):409–420, 1992.

[DJ89] Nachum Dershowitz and Jean-Pierre Jouannaud. *Rewrite systems.* Citeseer, 1989. 02335.

[ERKO11] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. SugarJ: library-based syntactic language extensibility. In *ACM SIGPLAN Notices*, volume 46, pages 391–406. ACM, 2011.

[GKS91] John RW Glauert, J Richard Kennaway, and M Ronan Sleep. Dactl: An experimental graph rewriting language. In *Graph Grammars and Their Application to Computer Science*, pages 378–395. Springer, 1991.

[Gra09] Albert Graf. *The Pure programming language.* Albert Graf, 2009. 00008.

[JTH01] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell workshop*, volume 1, pages 203–233, 2001.

[KB83] Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In *Automation of Reasoning*, pages 342–376. Springer, 1983.

[KDV90] Jan Willem Klop and R. C. De Vrijer. *Term rewriting systems.* Centrum voor Wiskunde en Informatica, 1990.

[Mes92] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical computer science*, 96(1):73–155, 1992.

[Pel11] Stephen Pelc. Programming forth. *Microprocessor Engineering Limited*, 2011.

[RO10] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Acm Sigplan Notices*, volume 46, pages 127–136. ACM, 2010.

[Sew98] Peter Sewell. *From rewrite to bisimulation congruences.* Springer, 1998. 00081.

[Sip12] Michael Sipser. *Introduction to the Theory of Computation.* Cengage Learning, 2012.

[Tah99] Walid Taha. *Multi-stage programming: Its theory and applications.* PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.

[X15] Sawyer X. perlref - search.cpan.org, February 2015.

A Program traces

```
"1 + 2 * 3 + 4 * 5"
"1 + 2 * 3" + "4 * 5"
("1" + "2 * 3") + "4 * 5"
(#<"1">#  + "2 * 3") + "4 * 5"
(1 + "2 * 3") + "4 * 5"
(1 + ("2" * "3")) + "4 * 5"
(1 + (#<"2">#  * "3")) + "4 * 5"
(1 + (2 * "3")) + "4 * 5"
(1 + (2 * #<"3">#)) + "4 * 5"
(1 + (2 * 3)) + "4 * 5"
(1 + (2 * 3)) + ("4" * "5")
(1 + (2 * 3)) + (#<"4">#  * "5")
(1 + (2 * 3)) + (4 * "5")
(1 + (2 * 3)) + (4 * #<"5">#)
(1 + (2 * 3)) + (4 * 5)
```

Figure 16: Parsing a regular arithmetic expression.

```
2 ^ 7
##* 2 (2 ^ (##- 7 1))
##* 2 (2 ^ 6)
##* 2 ((2 ^ (##div 6 2)) ^ 2)
##* 2 ((2 ^ 3) ^ 2)
##* 2 ((##* 2 (2 ^ (##- 3 1))) ^ 2)
##* 2 ((##* 2 (2 ^ 2)) ^ 2)
##* 2 ((##* 2 (##* 2 2)) ^ 2)
##* 2 ((##* 2 4) ^ 2)
##* 2 (8 ^ 2)
##* 2 (##* 8 8)
##* 2 64
128
```

Figure 17: Evaluating an integer exponent.

```
_a fifth --> a ^ 5
_a fifth --> ##* a (a ^ (##- 5 1))
_a fifth --> ##* a (a ^ 4)
_a fifth --> ##* a ((a ^ (##div 4 2)) ^ 2)
_a fifth --> ##* a ((a ^ 2) ^ 2)
_a fifth --> ##* a ((##* a a) ^ 2)
_a fifth --> ##* a (##* (##* a a) (##* a a))
```

Figure 18: Specializing a quintic integer power function.

```
recognize "() (())"
loop "() (())" 0
loop ") (())" (##+ 0 1)
loop ") (())" 1
loop "(())" (##- 1 1)
loop "(())" 0
loop "())" (##+ 0 1)
loop "())" 1
loop "))" (##+ 1 1)
loop "))" 2
loop ")" (##- 2 1)
loop ")" 1
loop "" (##- 1 1)
loop "" 0
accept
```

Figure 19: Recognizing balanced parentheses.

```
parse "1 * (2 + 3)"
loop "1 * (2 + 3)" $
loop " * (2 + 3)" ((num #<"1">#) $)
loop " * (2 + 3)" ((num 1) $)
loop " * (2 + 3)" ((trm 1) $)
loop " * (2 + 3)" ((fac 1) $)
loop "(2 + 3)" (* ((fac 1) $))
loop "2 + 3)" ([ (* ((fac 1) $)))
loop " + 3)" ((num #<"2">#) ([ (* ((fac 1) $))))
loop " + 3)" ((num 2) ([ (* ((fac 1) $))))
loop " + 3)" ((trm 2) ([ (* ((fac 1) $))))
loop " + 3)" ((fac 2) ([ (* ((fac 1) $))))
loop " + 3)" ((exp 2) ([ (* ((fac 1) $))))
loop "3)" (+ ((exp 2) ([ (* ((fac 1) $)))))
loop ")" ((num #<"3">#) (+ ((exp 2) ([ (* ((fac 1) $))))))
loop ")" ((num 3) (+ ((exp 2) ([ (* ((fac 1) $))))))
loop ")" ((trm 3) (+ ((exp 2) ([ (* ((fac 1) $))))))
loop ")" ((fac 3) (+ ((exp 2) ([ (* ((fac 1) $))))))
loop ")" ((exp (2 + 3)) ([ (* ((fac 1) $))))
loop "" (] ((exp (2 + 3)) ([ (* ((fac 1) $)))))
loop "" ((trm (2 + 3)) (* ((fac 1) $)))
loop "" ((fac (1 * (2 + 3))) $)
loop "" ((exp (1 * (2 + 3))) $)
accept (1 * (2 + 3))
```

Figure 20: Parsing a parenthesized arithmetic expression.