NORTHWESTERN UNIVERSITY

Automated Testing for Operational Semantics

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Electrical Engineering and Computer Science

By

Burke Fetscher

EVANSTON, ILLINOIS

December 2015

ABSTRACT

Automated Testing for Operational Semantics

Burke Fetscher

We present a technique for the random generation of well-typed expressions. The technique requires only the definition of a type-system in the form of inference rules and auxiliary functions, and produces random expressions satisfying the type system. In addition, we detail the implementation of a generator using this approach as part PLT Redex, a lightweight semantics modeling language and toolbox embedded in Racket. Specifically, we discuss a specialized constraint solver we have developed to support the generation of random derivations, and how Redex definitions are compiled into inputs for the solver during the process of generating a random type derivation.

Since our motivation for developing this generator is to do a better job at random testing, we also evaluate its random testing performance. To do so, we have developed a random-testing benchmark of Redex models. We discuss the development of the benchmark and show that our new approach to

Table of Contents

CHAPTER 1

Random Testing in PLT Redex

We begin with brief tour of the basics of Redex and the approach to semantics that it is chiefly designed to support in Section 2.1. Section 2.2 describes the facilities available for random testing in Redex as they existed prior to the present work, contrasting their demonstrated effectiveness and their shortcomings.

1.1. Overview of PLT Redex and Reduction Semantics

Redex is a domain-specific language for modelling programming languages. Users of Redex construct a model of a programming language using a notation that mimics as closely as possible the style used naturally by working language engineers in their papers. Redex models are executable and come with facilities for testing, debugging, and typesetting. Redex is embedded in Racket, so the full power of the Racket language, libraries and development infrastructure are available to to the Redex user.

Programming languages are modeled in Redex using *reduction semantics*, a concise form of operational semantics that is widely used. An operational semantics imparts meaning to a language by defining how computation takes place in that language. This is often done in terms of transformations on the syntax of the language. Reduction semantics uses a notion of contexts to define where and when computational steps may take place. In this section we demonstrate how reduction semantics works with a small example, and show how this example may be formalized using Redex. We pay special attention to details that will be important later in this dissertation.

Grammar

$$
\begin{array}{rl}
e ::= & .... \\
& \mid (\mathsf{if0}\ e\ e\ e) \\
& \mid (+\ e\ e) \\
v ::= & (\lambda\ (x\ \tau)\ e) \mid n \\
E ::= & (E\ e) \mid (v\ E) \mid (+\ E\ e) \mid (+\ v\ E) \\
& \mid (\mathsf{if0}\ E\ e\ e) \mid [\,]
\end{array}
$$

Reduction relation

$$E[((\lambda\ (x\ \tau)\ e)\ v)] \longrightarrow E[e\{x \leftarrow v\}] \quad [\beta]$$

$$E[(\mathsf{if0}\ 0\ e_1\ e_2)] \longrightarrow E[e_1] \quad [\text{if-0}]$$

$$E[(\mathsf{if0}\ n\ e_1\ e_2)] \longrightarrow E[e_2] \quad \text{where } n \neq 0 \quad [\text{if-n}]$$

$$E[(+\ n_1\ n_2)] \longrightarrow E[\lceil n_1 + n_2 \rceil] \quad [\text{plus}]$$

Metafunctions

$$
\begin{array}{rl}
\mathsf{lookup}[\![(x\ \tau\ \Gamma), x]\!] & = \tau \\
\mathsf{lookup}[\![(x_1\ \tau\ \Gamma), x_2]\!] & = \mathsf{lookup}[\![\Gamma, x_2]\!] \\
\mathsf{lookup}[\![\bullet, x]\!] & = \#\mathrm{f}
\end{array}
$$

Type judgment

$$\frac{}{\Gamma \vdash n : \mathsf{num}}$$

$$\frac{\tau = \mathsf{lookup}[\![\Gamma, x]\!]}{\Gamma \vdash x : \tau}$$

$$\frac{(x\ \tau_x\ \Gamma) \vdash e : \tau_e}{\Gamma \vdash (\lambda\ (x\ \tau_x)\ e) : (\tau_x \rightarrow \tau_e)}$$

$$\frac{\Gamma \vdash e_1 : (\tau_2 \rightarrow \tau) \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1\ e_2) : \tau}$$

$$\frac{\Gamma \vdash e_0 : \mathsf{num} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (\mathsf{if0}\ e_0\ e_1\ e_2) : \tau}$$

$$\frac{\Gamma \vdash e_0 : \mathsf{num} \qquad \Gamma \vdash e_1 : \mathsf{num}}{\Gamma \vdash (+\ e_0\ e_1) : \mathsf{num}}$$

Evaluation

$$\mathit{Eval}[\![e]\!] = n \quad \text{where } e \longrightarrow^* n$$

$$\mathit{Eval}[\![e]\!] = \mathsf{function} \quad \text{where } e \longrightarrow^* e_2, (\lambda\ (x\ \tau)\ e_3) = e_2$$

Figure 1: A reduction semantics and type system for the simply-typed lambda calculus.

A thorough introduction to reduction semantics can be found in Felleisen et al. (2010); they were first introduced in Felleisen and Hieb (1992).

Figure 1 shows a reduction semantics for the simply-typed lambda calculus. The lambda calculus (Barendregt 1984) models computation through function definition (abstraction) and application and is the basis of functional programming languages. The untyped lambda calculus, with an appropriate evaluation strategy[1] and some extensions, is a model of the core of Scheme, whereas the typed lambda calculus, as in this example, is the basis of languages such as ML. The grammar in the upper left of figure 1 delineates what valid terms in this language may look like. The e non-terminal describes the shapes of valid expressions. The v non-terminal denotes values, a special category of expressions that we are regarding as valid "answers" in this system — in this case they are lambda expressions (functions), or n's. Here n is used as a shorthand for numbers, and x as a shorthand for variables. The E non-terminal describes contexts, used in the reduction relation, and τ and Γ respectively describe types and the type environment, used in the type judgment.

The grammar of figure 1 can be expressed in Redex as:[2]

```
#f
```

The `define-language` form is used to describe a grammar and is usually the first element in a Redex program, since most other operations in Redex take place with respect to some language. It defines the language and binds it to an identifier, in this case our language is bound to `STLC`. After the language identifier comes a list of nonterminal definitions, each of which means that the non-terminal (left of `::=`) may be satisfied by some set of patterns (right of `::=`). Patterns are lists built from Racket constants, built-in Redex-patterns, or non-terminals. Here `number` and `variable-not-otherwise-mentioned` are built-in Redex patterns that respectively match Racket numbers and symbols that do not appear in productions in the grammar (λ is excluded, for example). Once a

[1] Such a strategy is defined here through contexts and the reduction relation.

[2] In fact, figure 1 is generated using Redex's typesetting facilities from precisely the code seen here, and the same is true for all the code corresponding to figure 1 in this section.

language is defined, the `redex-match` form can be used to ask Redex to attempt to match a pattern of literals and non-terminals against some term:

```
> (redex-match STLC (e_1 e_2)
                    (term ((λ (x num) x) 5)))
(list
 (match (list (bind 'e_1 '(λ (x num) x)) (bind 'e_2 5))))
```

Here `(e_1 e_2)` is the pattern we're attempting to match, a two-element list of terms that parse as `e`'s according to the grammar. A nonterminal followed by an underscore denotes a distinguished instance of that nonterminal, or a pattern metavariable. `term` is Redex's syntax for constructing s-expressions, in this case it is equivalent to Racket's `quote`. The result is a list of matches, each of which is a list of bindings for non-terminals in the pattern supplied to `redex-match`, and tells us what values `e_1` and `e_2` take for the match to succeed. In this case, only one match is possible, although in general there may be more.

The rest of the grammar is straightforward, with the exception of contexts, represented by the *E* non-terminal in this language. A context is distinguished by the fact that one of its productions is `[]`, denoting a "hole", which is a placeholder for some subexpression. The bulit-in Redex pattern `hole` is used to capture the notion of contexts, and is typeset as "`[]`". In patterns, contexts are used to decompose a term into the part outside the hole and the part inside the hole, and decomposed terms can be reassembled by "plugging" the hole with some new term. The notation *E[n]* is a pattern that matches some E context with a number in the hole, and *E[5]* would plug the hole and replace that number with *5*.

The `redex-match` form is useful for experimenting with contexts:

```
> (redex-match STLC (in-hole E (+ n_1 n_2))
```

```
                (term ((λ (x num) x) (+ 1 2))))
  (list
   (match
    (list
     (bind 'E '((λ (x num) x) hole))
     (bind 'n_1 1)
     (bind 'n_2 2))))
```

Redex's equivalent to the *E[]* notation is `in-hole`. The result indicates that this term can be successfully decomposed into a context containing an addition expression. Redex also allows us to experiment with decomposition and plugging:

```
> (redex-let STLC ([(in-hole E (+ n_1 n_2))
                    (term ((λ (x num) x) (+ 1 2)))])
      (term (in-hole E ,(+ (term n_1) (term n_2)))))
'((λ (x num) x) 3)
```

The decomposition is identical to the previous example, but `redex-let` binds the matches for use in `term`, where in this case the addition expression is replaced with an appropriate result. This example also demonstrates how `term` behaves similarly to Racket's `quasiquote`; the comma escapes out to Racket so we can use its "+" function.

The reduction relation, pictured below the grammar in figure 1, describes how computation takes place. The relation is described as a set of reduction rules, each of which matches a specific scheme of expressions on the left hand side of the arrow, and performs some transformation to a new scheme on the right hand side. Each rule is meant to describe a single step of computation. The left hand side of a rule is a *pattern*, which attempts to parse a *term* according to the grammar,

and if it succeeds, the non-terminals of the pattern are bound for use in the right hand to reconstruct a new term. Thus the relation pairs terms which match the left hand side to terms constructed by the right hand side, given the bound non-terminals from the match.

For example, ignoring reduction contexts for the moment, the rule for "+" is particularly simple — it simply transforms the addition expression of two numbers to the number that is their sum, exactly like the decompose/plug example above. The other rules describe function application by the substitution of the argument for the function's parameter (this rule is known as β-reduction)[3], and replace an "if" expression by one of its branches depending on the value of the test expression. The reduction of a term according to this relation describes computation in this language, and the interaction of the structure of the context and the form of the reduction relation together determine the order of reduction and computation.

The reduction relation can be formalized in Redex as:

```
(define STLC-red
  (reduction-relation STLC
   (--> (in-hole E ((λ (x τ) e) v))
        (in-hole E (subst e x v))
        β)
   (--> (in-hole E (if0 0 e_1 e_2))
        (in-hole E e_1)
        if-0)
   (--> (in-hole E (if0 n e_1 e_2))
        (in-hole E e_2)
```

[3]For simplicity, discussion of the subtle aspects of capture-avoiding substitution in the lambda calculus is omitted.

```
        (side-condition (term (different n 0)))
        if-n)
   (--> (in-hole E (+ n_1 n_2))
        (in-hole E (sum n_1 n_2))
        plus)))
```

Each `-->` takes a pattern as a first argument and a term (with the bindings from the pattern) as a second argument to create a reduction rule, and the relation itself is simply the union of all the rules.

The `apply-reduction-relation` form allows the Redex user to interactively explore the operation of the reduction relation on arbitrary terms. For example:

```
> (apply-reduction-relation
   STLC-red
   (term ((λ (x num) x) (+ 1 2))))
'(((λ (x num) x) 3))
```

Where the structure of the context E and the form of the rules determine where rules may be applied, and which rules may be applied. Here the rule for `+` reduces `(+ 1 2)` to `3` in an E context of the form `((λ (x num) x) [])`. Note that the result is actually a list of values, since in general an expression may be reduced in several ways by a reduction relation (in this case there is only one reduct).

It is useful to describe computation in terms of individual steps for accuracy, but ultimately a semantics is interested in the final result of evaluating and expression, or an answer. This is expressed by taking the transitive-reflexive closure of the reduction relation, denoted by $\rightarrow$*. If the transitive-reflexive closure of the reduction relates an expression e_1 to an expression e_2 that is not

in the domain of the reduction relation (does not take a step) then we say that the e_1 reduces to e_2, or that e_2 is a normal form[4] of e_1. Redex allows users to reduce expressions completely through the `apply-reduction-relation*` form, which returns a list of the final terms in all non-terminating reduction paths originating from its argument (when this is possible).[5]

```
> (apply-reduction-relation*
   STLC-red
   (term ((λ (x num) x) (+ 1 2))))
'(3)
```

Here Redex tells us that `3` is the result of the only possible sequence of reductions:

$$((\lambda\ (x\ num)\ x)\ (+\ 1\ 2)) \rightarrow ((\lambda\ (x\ num)\ x)\ 3) \rightarrow 3$$

where the first rule applied is *plus*, and the second is β.

The *Eval* function (at the base of figure 1) captures a complete notion of expression evaluation. It says that if the transitive-reflexive closure of some expression contains a number, the result of evaluating that expression is a number. If it contains a function, the result is simply the token `function`, which corresponds to what Racket and most languages that include first-class functions return when the result of an expression is a function.

Of course, *Eval* is unfortunately not defined for all expressions that satisfy the e non-terminal of our grammar. The evaluation of some expressions may terminate in non-values (known as "stuck" states), and some evaluations may never terminate. Non-terminating expressions are useful, but expressions that terminate in non-values don't have a defined answer and should be considered

[4]Not all expression have normal forms, and it is usually desirable that normal forms of allowable expressions other properties (i.e., are values). We use a type system to address these issues, which is described shortly.
[5]Although not detailed here, Redex also has support for exploring the details of reduction graphically.

errors. We can apply a type system to eliminate expressions that result in stuck states.[6] Type systems are relations between expressions and *types*, which denote what kind of value an expression will evaluate to. Thus an expression that has a type will reduce to a value. Types in our language match the τ non-terminal, and can correspond to either numbers ("num" types) or functions ("arrow" types). The type system is delineated by the inference rules of figure 1, which inductively describe a ternary relation between type environments Γ, expressions, and types. The statements above the horizontal lines are premises of each inference, and that on the bottom is the conclusion. This type of relation is known as a "judgment form", since it is typically used to make a decision about some syntactic object. (See Harper (2012) for more on judgment forms and type systems.) Judgment forms are typically used in practice by starting with a conclusion we wish to prove and attempting to construct a derivation of that conclusion. In this case one might start with an expression and try to construct a derivation that relates it to some type in the empty environment.[7] A "well-typed" expression is precisely that, one that satisfies the type judgment for some type in an empty environment, i.e. the expression e is well-typed if $\bullet \vdash e : \tau$, for some τ.

Type judgments and other judgment-forms are expressed in Redex using the `define-judgment-form` form. The following code, for example, implements the type system from figure 1:

```
(define-judgment-form STLC
  #:mode (tc I I O)
  [--------------
   (tc Γ n num)]
  [(where τ (lookup Γ x))
```

[6]The type system will necessarily also eliminate some expressions that don't result in stuck states, and in the case of the simple type system of this example, it also excludes non-terminating expressions.

[7]For this reason it is desirable that such judgments are "syntax directed", or that the form of the expression alone determines which rule applies in any case.

```
 ----------------------

 (tc Γ x τ)]

[(tc (x τ_x Γ) e τ_e)

 ---------------------------------

 (tc Γ (λ (x τ_x) e) (τ_x → τ_e))]

[(tc Γ e_1 (τ_2 → τ)) (tc Γ e_2 τ_2)

 ----------------------------------

 (tc Γ (e_1 e_2) τ)]

[(tc Γ e_0 num)

 (tc Γ e_1 τ) (tc Γ e_2 τ)

 ---------------------------

 (tc Γ (if0 e_0 e_1 e_2) τ)]

[(tc Γ e_0 num) (tc Γ e_1 num)

 ----------------------------

 (tc Γ (+ e_0 e_1) num)])
```

This corresponds closely to what is shown in figure 1, with the exception of the `mode` keyword argument. Redex requires that users provide a mode specification (Harper 2012) for a judgment form, which indicates how one or more of its arguments may be determined by other arguments. In this case, the mode indicates that the type environment and expression determine the type. Internally, Redex treats `I` mode positions as patterns and `O` positions as terms in the conclusion of a judgment (and the reverse in the premises). Thus a user asks Redex if a judgment form can be satisfied by providing terms for all the `I` positions, and Redex attempts to construct terms for the `O`

positions by matching against patterns in the conclusions and recursively calling the judgment in the premises. This is done using the `judgment-holds` form:

```
> (judgment-holds
    (tc • ((λ (x num) x) (+ 1 2)) τ))
#t
> (judgment-holds
    (tc • (+ (λ (x num) x) 2) τ))
#f
```

Here we ask Redex if the `tc` judgment can be satisfied for any type with the empty type environment and two expressions. (If necessary, Redex can also suppply the resulting type or types.) The first expression is well-typed (we have already seen that it reduces to a value), and the second is not (and would be a stuck state for the reduction).

Depicted below the reduction relation in figure 1, `lookup` is a *metafunction* used in the typing judgment. While a reduction relation may relate a term to multiple other terms if it matches the left hand side of multiple rules (or the same rule in multiple ways), a metafunction attempts to match the left hand sides of its cases in order, and the input term is mapped to the right hand side of the first successful match.[8] The pattern matching and term construction processes are similar to the reduction relation. Metafunctions are expressed in Redex using the `define-metafunction` form:

```
(define-metafunction STLC
  [(lookup (x τ Γ) x)
   τ]
  [(lookup (x_1 τ Γ) x_2)
```

[8]Redex considers multiple matches for the left-hand side of a single metafunction clause an error.

```
   (lookup Γ x_2)]
  [(lookup • x)
   #f])
```

The lookup metafunction takes a variable x and an enviroment Γ and first tries to match x to the beginning of the environment, returning the corresponding type τ if it succeeds. Otherwise, it recurs on what remains of Γ. The final case indicates that lookup fails and returns *#f*, or *false*, when passed an empty environment. Metafunctions are functions on and in the object language but are defined in the meta-language (the language of the semantics: in this case, Redex) – in Redex, metafunctions may only be used inside of `term` (including implicit uses of `term`, such as the right-hand sides of reduction relations or metafunctions themselves). For example:

```
> (term (lookup (a num
                   (b (num → num)
                      (b num •)))
                b))
'(num → num)
> (term (lookup (a num
                   (b (num → num)
                      (b num •)))
                c))
#f
```

1.2. Random Testing in Redex

Prior to the present work, all random test case generation in Redex followed a remarkably simple strategy based solely on grammars. Given a Redex pattern and its corresponding grammar, the test case generator chooses randomly among the productions for each non-terminal in the pattern. If the non-terminal itself is a pattern, then the same strategy is used recursively. Since an unbounded use of this strategy is likely to produce arbitrarily large terms, the generator also receives a depth parameter which is decremented on recursive calls. Once the depth parameter reaches zero, the generator prefers non-recursive productions in the grammar. After some brief examples we will discuss the effectiveness of this strategy in the context of producing terms which satisfy some desirable property (such as well-typedness). Klein (2009) discusses the implementation, application, and inherent tradeoffs of this testing strategy in detail.

Test case generators can be called directly with the `generate-term` form, which take a language, a pattern, and a depth limit as parameters. To generate a random expression in the simply-typed lambda calculus model from the previous section:

```
> (generate-term STLC e 5)
'(z (if0 v N (λ (r (num → num)) (0 0))))
```

We can see that this term satisfies the grammar. We can also see that it has a number of free variables, so it is not well-typed and will result in a "stuck state" (a normal form that is not a value) for this reduction relation – a point we will return to momentarily.

First, let's examine how we might test our system in actuality using Redex's testing infrastructure. We may wish, for example, to test that expressions in our language do not result in stuck states. To do so, one might write a predicate checking for that property, and then generate a collection of terms and check that the predicate is never false. This is a common situation, so Redex

provides redex-check, a form that conducts such testing automatically, given a user-provided predicate. The following code checks a property asserting that terms always reduce to values:

```
> (redex-check STLC e
               (redex-match STLC v
                            (car (apply-reduction-relation*
                                  STLC-red
                                  (term e)))))
redex-check: counterexample found after 1 attempt:
a
```

The first two arguments tell redex-check to generate terms in the STLC language matching the pattern e (i.e. expressions). The third is the predicate that defines the property we wish to check; in this case, it says that taking the transitive-reflexive closure of the generated expression with respect to the reduction relation should produce a value. (For simplicity the assumption has been made that the reduction sequence always terminates in a single unique result, which happens to be true in this case but is not in general.) Here a counterexample, a free variable, is found quickly. Of course this is because in this case the property being tested is incorrect: we first need to check that the expression is a valid program, or that it is well typed. Here is the test adjusted accordingly:

```
> (redex-check STLC e
               (or (not (judgment-holds (tc • e τ)))
                   (redex-match STLC v
                                (car (apply-reduction-relation*
                                      STLC-red
```

```
                                        (term e))))))
redex-check: no counterexamples in 1000 attempts
```

And now Redex is unable to find any counterexamples in the default 1000 test cases. That is encouraging, but we may wonder how good our test coverage is. We can wrap the previous test with some code asking Redex to tell us how many times each rule in the reduction relation is exercised:

```
> (let ([red-coverage (make-coverage STLC-red)])
    (parameterize
        ([relation-coverage (list red-coverage)])
      (redex-check STLC e
                   (or (not (judgment-holds (tc • e τ)))
                       (redex-match STLC v
                                    (car (apply-reduction-relation*
                                          STLC-red
                                          (term e))))))
      (covered-cases red-coverage)))
redex-check: no counterexamples in 1000 attempts
'(("if-0" . 0) ("if-n" . 2) ("plus" . 4) ("β" . 4))
```

The result reports that the "plus" reduction rule was used 3 times over the 1000 test cases, and none of the other rules was used even once. Clearly, the vast majority of the terms being generated are poor tests. To give an idea of what kind of terms are generated, figure 2 shows some statistics for random terms in this language, and exposes some of the difficulty inherent in generating "good"

Term characteristic	**Percentage of Terms**
Well-typed	11.36%
Reduces once or more	7.03%
Uses if-else rule	3.03%
Uses if-0 rule	2.88%
Uses β rule	0.95%
Uses plus rule	0.70%
Reduces twice or more	0.69%
Well-typed, not a constant or constant function	0.51%
Well-typed, reduces once or more	0.26%
Reduces three or more times	0.07%

Figure 2: Statistics for 100000 terms randomly generated from the stlc grammar.

terms. Although about 10% of random expresssions are well typed, only 0.5% are well-typed and not a constant or a constant function (a function of the form *(λ (x τ) n)*). The terms that are good tests for the property in question, those that are well-typed and exercise the reduction, are even rarer, at 0.26% of all terms. Thus it is unsurprising that test coverage is so poor.

The use of a few basic strategies can significantly improve the coverage of terms generated using this method. Redex can generate terms using the patterns of the left-hand-sides of the reduction rules as templates, which increases the chances of generating a term exercising each case. However, it is still likely that such terms will fail to be well-typed. Frequently this is due to the presence of free variables in the term. Thus the user can write a function to preprocess randomly generated terms by attempting to bind free variables. Both approaches are well-supported by `redex-check`.

The strategy of using strategically selected source patterns and preprocessing terms in some way is typical of most serious testing efforts involving Redex, and has been effective in many cases. It has been used to successfully find bugs in a Scheme semantics (Klein 2009), the Racket Virtual Machine (Klein et al. 2013), and various language models drawn from the International Conference on Functional Programming (Klein et al. 2012). However, given the apparent rarity of

useful terms even in a language as simple as our small lambda calculus example, one may wonder if there is not a more effective term generation strategy that requires less work and ingenuity from the user.

The remainder of this dissertation details work on the approach of attempting to generate terms directly from Redex judgment-forms and metafunctions, since those terms will in many cases inherently have desirable properties from a testing standpoint. We then attempt to compare the effectiveness of this strategy to the one explained in this section.

CHAPTER 2

Derivation Generation in Detail

This section describes a formal model[1] of the derivation generator. The centerpiece of the model is a relation that rewrites programs consisting of metafunctions and judgment forms into the set of possible derivations that they can generate. Our implementation has a structure similar to the model, except that it uses randomness and heuristics to select just one of the possible derivations that the rewriting relation can produce. Our model is based on Jaffar et al. (1998)'s constraint logic programming semantics.

[1]The corresponding Redex model is available from this paper's website (listed after the conclusion), including a runnable simple example that may prove helpful when reading this section.

Programs	Formulas	Patterns
$P ::= (D \ ...)$	$C ::= (\wedge\ (\wedge\ (x = p)\ ...)$	$p ::= (\mathsf{lst}\ p\ ...)$
$D ::= (r \ ...)$	$\quad (\wedge\ \delta\ ...))$	$\quad \mid m$
$r ::= ((d\ p) \leftarrow a\ ...)$	$e ::= (p = p)$	$\quad \mid x$
$a ::= (d\ p) \mid \delta$	$\delta ::= (\forall\ (x\ ...)\ (\vee\ (p \neq p)\ ...))$	$m ::= \mathit{Constant}$
$d ::= \mathit{Identifier}$		$x ::= \mathit{Variable}$

Figure 3: The syntax of the derivation generator model.

$$
\begin{array}{l}
(P \vdash ((d\ p_g)\ a\ ...) \parallel C_1) \qquad\qquad \text{[reduce]} \\
\longrightarrow (P \vdash (a_f\ ...\ a\ ...) \parallel C_2) \\
\quad \text{where } (D_0\ ...\ (r_0\ ...\ ((d\ p_r) \leftarrow a_r\ ...)\ r_1\ ...)\ D_1\ ...) = P, \\
\qquad ((d\ p_f) \leftarrow a_f\ ...) = \mathsf{freshen}[\![((d\ p_r) \leftarrow a_r\ ...)]\!], \\
\qquad C_2 = \mathsf{solve}[\![(p_f = p_g), C_1]\!] \\
\\
(P \vdash (\delta_g\ a\ ...) \parallel C_1) \qquad\qquad \text{[new constraint]} \\
\longrightarrow (P \vdash (a\ ...) \parallel C_2) \\
\quad \text{where } C_2 = \mathsf{dissolve}[\![\delta_g, C_1]\!]
\end{array}
$$

Figure 4: Reduction rules describing generation of the complete tree of derivations.

The grammar in figure 3 describes the language of the model. A program P consists of definitions D, which are sets of inference rules $((d\ p) \leftarrow a\ ...)$, here written horizontally with the conclusion on the left and premises on the right. (Note that ellipses are used in a precise manner to indicate repetition of the immediately previous expression, in this case a, following Scheme tradition. They do not indicate elided text.) Definitions can express both judgment forms and metafunctions. They are a strict generalization of judgment forms, and metafunctions are compiled into them via a process we discuss in section 2.1.

The conclusion of each rule has the form $(d\ p)$, where d is an identifier naming the definition and p is a pattern. The premises a may consist of literal goals $(d\ p)$ or disequational constraints δ. We dive into the operational meaning behind disequational constraints later in this section, but as their form in figure 3 suggests, they are a disjunction of negated equations, in which the variables listed following $\forall$ are universally quantified. The remaining variables in a disequation are implicitly existentially quantified, as are the variables in equations.

The reduction relation shown in figure 4 generates the complete tree of derivations for the program P with an initial goal of the form $(d\ p)$, where d is the identifier of some definition in P and p is a pattern that matches the conclusion of all of the generated derivations. The relation is defined using two rules: [reduce] and [new constraint]. The states that the relation acts on are of the form $(P \vdash (a\ ...) \parallel C)$, where $(a\ ...)$ represents a stack of goals, which can either be incomplete derivations of the form $(d\ p)$, indicating a goal that must be satisfied to complete the derivation, or disequational constraints that must be satisfied. A constraint store C is a set of simplified equations and disequations that are guaranteed to be satisfiable. The notion of equality we use here is purely syntactic; two ground terms are equal to each other only if they are identical.

Each step of the rewriting relation looks at the first entry in the goal stack and rewrites to another state based on its contents. In general, some reduction sequences are ultimately doomed, but may still reduce for a while before the constraint store becomes inconsistent. In our implementation, discovery of such doomed reduction sequences causes backtracking. Reduction sequences that lead to valid derivations always end with a state of the form $(P \vdash () \parallel C)$, and the derivation itself can be read off of the reduction sequence that reaches that state.

When a goal of the form $(d\ p)$ is the first element of the goal stack (as is the root case, when the initial goal is the sole element), then the [reduce] rule applies. For every rule of the form $((d\ p_r) \leftarrow a_r\ ...)$ in the program such that the definition's id d agrees with the goal's, a reduction step can occur. The reduction step first freshens the variables in the rule, asks the solver to combine the equation $(p_f = p_g)$ with the current constraint store, and reduces to a new state with the new constraint store and a new goal state. If the solver fails, then the reduction rule doesn't apply (because solve returns $\perp$ instead of a C_2). The new goal stack has all of the previously pending goals as well as the new ones introduced by the premises of the rule.

The [new constraint] rule covers the case where a disequational constraint δ is the first element in the goal stack. In that case, the disequational solver is called with the current constraint store and the disequation. If it returns a new constraint store, then the disequation is consistent and the new constraint store is used.

The remainder of this section fills in the details in this model and discusses the correspondence between the model and the implementation in more detail. Metafunctions are added via a procedure generalizing the process used for lookup in section ???, which we explain in section 2.1. Section 2.2 describes how our solver handles equations and disequations. Section 2.3 discusses the heuristics in our implementation and section 2.4 describes how our implementation scales up to support features in Redex that are not covered in this model.

2.1. Compiling Metafunctions

The primary difference between a metafunction, as written in Redex, and a set of $((d\ p) \leftarrow a\ ...)$ clauses from figure 3 is sensitivity to the ordering of clauses. Specifically, when the second clause in a metafunction fires, then the pattern in the first clause must not match, in contrast to the rules in the model, which fire regardless of their relative order. Accordingly, the compilation process that translates metafunctions into the model must insert disequational constraints to capture the ordering of the cases.

As an example, consider the metafunction definition of g on the left and some example applications on the right:

$$\begin{array}{lll} \mathsf{g}[\![(\mathsf{lst}\ p_1\ p_2)]\!] = 2 & & \mathsf{g}[\![(\mathsf{lst}\ 1\ 2)]\!] = 2 \\ \mathsf{g}[\![p]\!] = 1 & & \mathsf{g}[\![(\mathsf{lst}\ 1\ 2\ 3)]\!] = 1 \end{array}$$

The first clause matches any two-element list, and the second clause matches any pattern at all. Since the clauses apply in order, an application where the argument is a two-element list will reduce to 2 and an argument of any other form will reduce to 1. To generate conclusions of the

judgment corresponding to the second clause, we have to be careful not to generate anything that matches the first.

Applying the same idea as lookup in section ???, we reach this incorrect translation:

$$\frac{}{\mathsf{g}[\![(\mathsf{lst}\ p_1\ p_2)]\!] = 2} \qquad \frac{(\mathsf{lst}\ p_1\ p_2) \neq p}{\mathsf{g}[\![p]\!] = 1}$$

This is wrong because it would let us derive $\mathsf{g}[\![(\mathsf{list}\ 1\ 2)]\!] = 1$, using 3 for p_1 and 4 for p_2 in the premise of the right-hand rule. The problem is that we need to disallow all possible instantiations of p_1 and p_2, but the variables can be filled in with just specific values to satisfy the premise.

The correct translation, then, universally quantifies the variables p_1 and p_2:

$$\frac{}{\mathsf{g}[\![(\mathsf{lst}\ p_1\ p_2)]\!] = 2} \qquad \frac{(\forall(p_1\ p_2)\ (\mathsf{lst}\ p_1\ p_2) \neq p)}{\mathsf{g}[\![p]\!] = 1}$$

Thus, when we choose the second rule, we know that the argument will never be able to match the first clause.

In general, when compiling a metafunction clause, we add a disequational constraint for each previous clause in the metafunction definition. Each disequality is between the left-hand side patterns of one of the previous clauses and the left-hand side of the current clause, and it is quantified over all variables in the previous clause's left-hand side.

2.2. The Constraint Solver

The constraint solver maintains a set of equations and disequations that captures invariants of the current derivation that it is building. These constraints are called the constraint store and are kept in the canonical form C, as shown in figure 3, with the additional constraint that the equational portion of the store can be considered an idempotent substitution. That is, it always equates variables with with ps and, no variable on the left-hand side of an equality also appears

in any right-hand side. Whenever a new constraint is added, consistency is checked again and the new set is simplified to maintain the canonical form.

Figure 5 shows solve, the entry point to the solver for new equational constraints. It accepts an equation and a constraint store and either returns a new constraint store that is equivalent to the conjunction of the constraint store and the equation or $\bot$, indicating that adding e is inconsistent with the constraint store. In its body, it first applies the equational portion of the constraint store as a substitution to the equation. Second, it performs syntactic unification (Baader and Snyder 2001) of the resulting equation with the equations from the original store to build a new equational portion of the constraint. Third, it calls check, which simplifies the disequational constraints and checks their consistency. Finally, if all that succeeds, check returns a constraint store that combines the results of unify and check. If either unify or check fails, then solve returns $\bot$.

Figure 6 shows dissolve, the disequational counterpart to solve. It applies the equational part of the constraint store as a substitution to the new disequation and then calls disunify. It disunify returns $\top$, then the disequation was already guaranteed in the current constraint store and thus does not need to be recorded. If disunify returns $\bot$ then the disequation is inconsistent with the current constraint store and thus dissolve itself returns $\bot$. In the final situation, disunify returns a new disequation, in which case dissolve adds that to the resulting constraint store.

The disunify function exploits unification and a few cleanup steps to determine if the input disequation is satisfiable. In addition, disunify is always called with a disequation that has had the equational portion of the constraint store applied to it (as a substitution).

The key trick in this function is to observe that since a disequation is always a disjunction of inequalities, its negation is a conjuction of equalities and is thus suitable as an input to unification. The first case in disunify covers the case where unification fails. In this situation we know that the disequation must have already been guaranteed to be false in constraint store (since the equational

$$\mathsf{solve} : e\ C \to C \text{ or } \bot$$

$$\mathsf{solve}[\![e_{new}, (\wedge\ (\wedge\ (x = p)\ ...)\ (\wedge\ \delta\ ...))]\!] =$$

$$\begin{cases} (\wedge\ (\wedge\ (x_2 = p_2)\ ...) & \text{if } (\wedge\ (x_2 = p_2)\ ...) = \mathsf{unify}[\![(e_{new}\{x \to p, ...\}), (\wedge\ (x = p)\ ...)]\!], \\ \quad (\wedge\ \delta_2\ ...)) & \quad (\wedge\ \delta_2\ ...) = \mathsf{check}[\![(\wedge\ \delta\{x_2 \to p_2, ...\}\ ...)]\!] \\ \bot & \text{otherwise} \end{cases}$$

$$\mathsf{unify} : (e\ ...)\ (\wedge\ (x = p)\ ...) \to (\wedge\ (x = p)\ ...) \text{ or } \bot$$

$$\mathsf{unify}[\![((p = p)\ e\ ...), (\wedge\ e_s\ ...)]\!] = \mathsf{unify}[\![(e\ ...), (\wedge\ e_s\ ...)]\!]$$

$$\mathsf{unify}[\![(((\mathsf{lst}\ p_1\ ..._1) = (\mathsf{lst}\ p_2\ ..._1))\ e\ ...), (\wedge\ e_s\ ...)]\!] = \mathsf{unify}[\![((p_1 = p_2)\ ...\ e\ ...), (\wedge\ e_s\ ...)]\!]$$
where $|(p_{1...})| = |(p_2\ ...)|$

$$\mathsf{unify}[\![((x = p)\ e\ ...), (\wedge\ e_s\ ...)]\!] = \bot$$
where $\mathsf{occurs?}[\![x, p]\!], x \neq p$

$$\mathsf{unify}[\![((x = p)\ e\ ...), (\wedge\ e_s\ ...)]\!] = \mathsf{unify}[\![(e\{x \to p\}\ ...), (\wedge\ (x = p)\ e_s\{x \to p\}\ ...)]\!]$$

$$\mathsf{unify}[\![((p = x)\ e\ ...), (\wedge\ e_s\ ...)]\!] = \mathsf{unify}[\![((x = p)\ e\ ...), (\wedge\ e_s\ ...)]\!]$$

$$\mathsf{unify}[\![(), (\wedge\ e\ ...)]\!] = (\wedge\ e\ ...)$$

$$\mathsf{unify}[\![(e\ ...), (\wedge\ e_s\ ...)]\!] = \bot$$

Figure 5: The Solver for Equations

portion of the constraint store was applied as a substitution before calling disunify). Accordingly, disunify can simply return $\top$ to indicate that the disequation was redundant.

Ignoring the call to param-elim in the second case of disunify for a moment, consider the case where unify returns an empty conjunct. This means that unify's argument is guaranteed to be true and thus the given disequation is guaranteed to be false. In this case, we have failed to generate a valid derivation because one of the negated disequations must be false (in terms of the original Redex program, this means that we attempted to use some later case in a metafunction with an input that would have satisfied an earlier case) and so disunify must return $\bot$.

$$\textsf{dissolve} : \delta\ C \to C \text{ or } \bot$$

$$\textsf{dissolve}[\![\delta_{new}, (\wedge\ (\wedge\ (x = p)\ ...)\ (\wedge\ \delta\ ...))]\!] =$$

$$\begin{cases} (\wedge\ (\wedge\ (x = p)\ ...)\ (\wedge\ \delta\ ...)) & \text{if } \top = \textsf{disunify}[\![\delta_{new}\{x \to p, ...\}]\!] \\ \bot & \text{if } \bot = \textsf{disunify}[\![\delta_{new}\{x \to p, ...\}]\!] \\ (\wedge\ (\wedge\ (x = p)\ ...)\ (\wedge\ \delta_0\ \delta\ ...)) & \text{if } \delta_0 = \textsf{disunify}[\![\delta_{new}\{x \to p, ...\}]\!] \end{cases}$$

$$\textsf{disunify} : \delta \to \delta \text{ or } \top \text{ or } \bot$$

$$\textsf{disunify}[\![(\forall\ (x\ ...)\ (\vee\ (p_1 \neq p_2)\ ...))]\!] =$$

$$\begin{cases} \top & \text{if } \bot = \textsf{unify}[\![((p_1 = p_2)\ ...), (\wedge)]\!] \\ \bot & \text{if } (\wedge) = \textsf{param-elim}[\![\textsf{unify}[\![((p_1 = p_2)\ ...), (\wedge)]\!], (x\ ...)]\!] \\ (\forall\ (x\ ...)\ (\vee\ (x_p \neq p)\ ...)) & \text{if } (\wedge\ (x_p = p)\ ...) = \textsf{param-elim}[\![\textsf{unify}[\![((p_1 = p_2)\ ...), (\wedge)]\!], (x\ ...)]\!] \end{cases}$$

Figure 6: The Solver for Disequations

But there is a subtle point here. Imagine that unify returns only a single clause of the form $(x = p)$ where x is one of the universally quantified variables. We know that in that case, the corresponding disequation $(\forall\ (x)\ (x \neq p))$ is guaranteed to be false because every pattern admits at least one concrete term. This is where param-elim comes in. It cleans up the result of unify by eliminating all clauses that, when negated and placed back under the quantifier would be guaranteed false, so the reasoning in the previous paragraph holds and the second case of disunify behaves properly.

The last case in disunify covers the situation where unify composed with param-elim returns a non-empty substitution. In this case, we do not yet know if the disequation is true or false, so we collect the substitution that unify returned back into a disequation and return it, to be saved in the constraint store.

$$\textsf{check} : (\wedge\ \delta\ ...) \to (\wedge\ \delta\ ...) \text{ or } \bot$$

$$\textsf{check}[\![(\wedge\ \delta_1\ ...\ (\forall\ (x_a\ ...)\ (\vee\ ((\textsf{lst}\ p_l\ ...) \neq p_r)\ ...))\ \delta_2\ ...)]\!] =$$

$$\begin{cases} \textsf{check}[\![(\wedge\ \delta_1\ ...\ \delta_s\ \delta_2\ ...)]\!] & \text{if } \delta_s = \textsf{disunify}[\![(\forall\ (x_a\ ...)\ (\vee\ ((\textsf{lst}\ p_l\ ...) \neq p_r)\ ...))]\!] \\ \textsf{check}[\![(\wedge\ \delta_1\ ...\ \delta_2\ ...)]\!] & \text{if } \top = \textsf{disunify}[\![(\forall\ (x_a\ ...)\ (\vee\ ((\textsf{lst}\ p_l\ ...) \neq p_r)\ ...))]\!] \\ \bot & \text{if } \bot = \textsf{disunify}[\![(\forall\ (x_a\ ...)\ (\vee\ ((\textsf{lst}\ p_l\ ...) \neq p_r)\ ...))]\!] \end{cases}$$

$$\textsf{check}[\![(\wedge\ \delta\ ...)]\!] = (\wedge\ \delta\ ...)$$

$$\textsf{param-elim} : (\wedge\ e\ ...)\ (x\ ...) \to (\wedge\ e\ ...) \text{ or } \bot$$

$$\textsf{param-elim}[\![(\wedge\ (x_0 = p_0)\ ...\ (x = p)\ (x_1 = p_1)\ ...), (x_2\ ...\ x\ x_3\ ...)]\!] =$$
$$\quad \textsf{param-elim}[\![(\wedge\ (x_0 = p_0)\ ...\ (x_1 = p_1)\ ...), (x_2\ ...\ x\ x_3\ ...)]\!]$$

$$\textsf{param-elim}[\![(\wedge\ (x_0 = p_0)\ ...\ (x_1 = x)\ (x_2 = p_2)\ ...), (x_4\ ...\ x\ x_5\ ...)]\!] =$$
$$\quad \textsf{param-elim}[\![(\wedge\ (x_0 = p_0)\ ...\ (x_3 = p_3)\ \ ...), (x_4\ ...\ x\ x_5\ ...)]\!]$$
$$\text{where } x \notin (p_0\ ...), ((x_3 = p_3)\ ...) = \textsf{elim-x}[\![x, ((x_1 = x)\ (x_2 = p_2)\ ...), ()]\!]$$

$$\textsf{param-elim}[\![(\wedge\ e\ ...), (x\ ...)]\!] = (\wedge\ e\ ...)$$

$$\textsf{elim-x}[\![x, ((p_0 = p_1)\ ...\ (p_2 = x)\ e_2\ ...), (e_3\ ...)]\!] =$$
$$\quad \textsf{elim-x}[\![x, ((p_0 = p_1)\ ...\ \ e_2\ ...), (e_3\ ...\ (p_2 = x))]\!]$$
$$\text{where } x \notin (p_1\ ...)$$
$$\textsf{elim-x}[\![x, (e_1\ ...), ((p_2 = x_2)\ ...)]\!] = (e_1\ ...\ e_2\ ...)$$
$$\text{where } (e_2\ ...) = \textsf{all-pairs}[\![(p_2\ ...), ()]\!]$$

$$\textsf{all-pairs}[\![(p_1\ p_2\ ...), (e\ ...)]\!] = \textsf{all-pairs}[\![(p_2\ ...), (e\ ...\ (p_1 = p_2)\ ...)]\!]$$
$$\textsf{all-pairs}[\![(), (e\ ...)]\!] = (e\ ...)$$

Figure 7: Metafunctions used to process disequational constaints.

This brings us to param-elim, in figure 7. Its first argument is a unifier, as produced by a call to unify to handle a disequation, and the second argument is the universally quantified variables from the original disequation. Its goal is to clean up the unifier by removing redundant and useless clauses.

There are two ways in which clauses can be false. In addition to clauses of the form $(x = p)$ where x is one of the universally quantified variables, it may also be the case that we have a clause of the form $(x_1 = x)$ and, as before, x is one of the universally quantified variables. This clause also

must be dropped, according to the same reasoning (since = is symmetric). But, since variables on the right hand side of an equation may also appear elsewhere, some care must be taken here to avoid losing transitive inequalities. The function elim-x (not shown) handles this situation, constructing a new set of clauses without x but, in the case that we also have $(x_2 = x)$, adds back the equation $(x_1 = x_2)$. For the full definition of elim-x and a proof that it works correctly, we refer the reader to the first author's masters dissertation (Fetscher 2014).

Finally, we return to check, shown in figure 7, which is passed the updated disequations after a new equation has been added in solve (see figure 5). It verifies the disequations and maintains their canonical form, once the new substitution has been applied. It does this by applying disunify to any non-canonical disequations.

2.3. Search Heuristics

To pick a single derivation from the set of candidates, our implementation must make explicit choices when there are differing states that a single reduction state reduces to. Such choices happen only in the [reduce] rule, and only because there may be multiple different clauses, $((d\ p) \leftarrow a\ ...)$, that could be used to generate the next reduction state.

To make these choices, our implementation collects all of the candidate cases for the next definition to explore. It then randomly permutes the candidate rules and chooses the first one of the permuted rules, using it as the next piece of the derivation. It then continues to search for a complete derivation. That process may fail, in which case the implementation backtracks to this choice and picks the next rule in the permuted list. If none of the choices leads to a successful derivation, then this attempt is failure and the implementation either backtracks to an earlier such choice, or fails altogether.

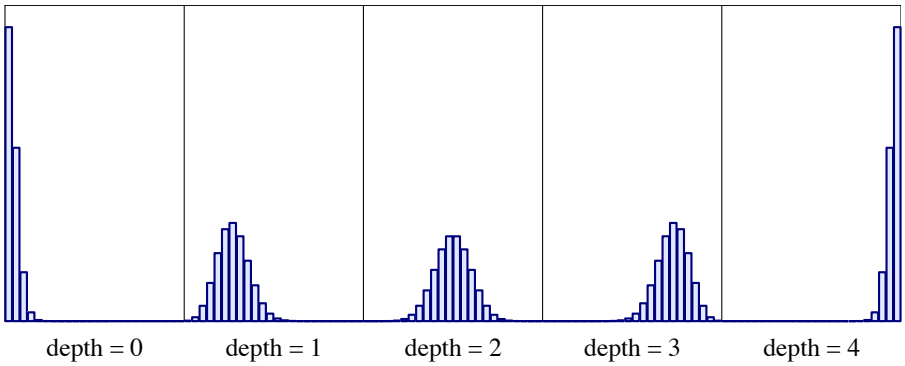

Figure 8: Density functions of the distributions used for the depth-dependent rule ordering, where the depth limit is 4 and there are 4 rules.

There are two refinements that the implementation applies to this basic strategy. First, the search process has a depth bound that it uses to control which production to choose. Each choice of a rule increments the depth bound and when the partial derivation exceeds the depth bound, then the search process no longer randomly permutes the candidates. Instead, it simply sorts them by the number of premises they have, preferring rules with fewer premises in an attempt to finish the derivation off quickly.

The second refinement is the choice of how to randomly permute the list of candidate rules, and the generator uses two strategies. The first strategy is to just select from the possible permutations uniformly at random. The second strategy is to take into account how many premises each rule has and to prefer rules with more premises near the beginning of the construction of the derivation and rules with fewer premises as the search gets closer to the depth bound. To do this, the implementation sorts all of the possible permutations in a lexicographic order based on the number of premises of each choice. Then, it samples from a binomial distribution whose size matches the number of permutations and has probability proportional to the ratio of the current depth and the maximum depth. The sample determines which permutation to use.

More concretely, imagine that the depth bound was 4 and there are also 4 rules available. Accordingly, there are 24 different ways to order the premises. The graphs in figure 8 show the probability of choosing each permutation at each depth. Each graph has one x-coordinate for each different permutation and the height of each bar is the chance of choosing that permutation. The permutations along the x-axis are ordered lexicographically based on the number of premises that each rule has (so permutations that put rules with more premises near the beginning of the list are on the left and permutations that put rules with more premises near the end of the list are on the right). As the graph shows, rules with more premises are usually tried first at depth 0 and rules with fewer premises are usually tried first as the depth reaches the depth bound.

These two permutation strategies are complementary, each with its own drawbacks. Consider using the first strategy that gives all rule ordering equal probability with the rules shown in figure #f. At the initial step of our derivation, we have a 1 in 4 chance of choosing the type rule for numbers, so one quarter of all expressions generated will just be a number. This bias towards numbers also occurs when trying to satisfy premises of the other, more recursive clauses, so the distribution is skewed toward smaller derivations, which contradicts commonly held wisdom that bug finding is more effective when using larger terms. The other strategy avoids this problem, biasing the generation towards rules with more premises early on in the search and thus tending to produce larger terms. Unfortunately, our experience testing Redex program suggests that it is not uncommon for there to be rules with large number of premises that are completely unsatisfiable when they are used as the first rule in a derivation (when this happens there are typically a few other, simpler rules that must be used first to populate an environment or a store before the interesting and complex rule can succeed). For such models, using all rules with equal probability still is less than ideal, but is overall more likely to produce terms at all.

```
p ::= (nt s)
    | (name s p)
    | (mismatch-name s p)
    | (list p ...)
    | b
    | v
    | c
v ::= variable
    | (variable-except s ...)
    | (variable-prefix s)
    | variable-not-otherwise-mentioned
b ::= any
    | number
    | string
    | natural
    | integer
    | real
    | boolean
s ::= symbol
c ::= constant
```

Figure 9: The subset of Redex's pattern language supported by the generator. Racket symbols are indicated by *s*, and *c* represents any Racket constant.

Since neither strategy for ordering rules is always better than the other, our implementation decides between the two randomly at the beginning of the search process for a single term, and uses the same strategy throughout that entire search. This is the approach the generator we evaluate in section ??? uses.

Finally, in all cases we terminate searches that appear to be stuck in unproductive or doomed parts of the search space by placing limits on backtracking, search depth, and a secondary, hard bound on derivation size. When these limits are violated, the generator simply abandons the current search and reports failure.

2.4. A Richer Pattern Language

The model we present in section 2 uses a much simpler pattern language than Redex itself. The portion of Redex's internal pattern language supported by the generator[2] is shown in figure 9. We now discuss briefly the interesting differences between this language and the language of our model and how we support them in Redex's implementation.

[2]The generator is not able to handle parts of the pattern language that deal with evaluation contexts or "repeat" patterns (ellipses).

Named patterns of the form (name *s p*) correspond to variables *x* in the simplified version of the pattern language from figure 3, except that the variable *s* is paired with a pattern *p*. From the matcher's perspective, this form is intended to match a term with the pattern *p* and then bind the matched term to the name *s*. The generator pre-processes all patterns with a first pass that extracts the attached pattern *p* and attempts to update the current constraint store with the equation ($s = p$), after which *s* can be treated as a logic variable.

The *b* and *v* non-terminals are built-in patterns that match subsets of Racket values. The productions of *b* are straightforward; integer, for example, matches any Racket integer, and any matches any Racket s-expression. From the perspective of the unifier, integer is a term that may be unified with any integer, the result of which is the integer itself. The value of the term in the current substitution is then updated. Unification of built-in patterns produces the expected results; for example unifying real and natural produces natural, whereas unifying real and string fails.

The productions of *v* match Racket symbols in varying and commonly useful ways; variable-not-otherwise-m for example, matches any symbol that is not used as a literal elsewhere in the language. These are handled similarly to the patterns of the *b* non-terminal within the unifier.

Patterns of the from (mismatch-name *s p*) match the pattern *p* with the constraint that two occurrences of the same name *s* may never match equal terms. These are straightforward: whenever a unification with a mismatch takes place, disequations are added between the pattern in question and other patterns that have been unified with the same mismatch pattern.

Patterns of the form (nt *s*) refer to a user-specified grammar, and match a term if it can be parsed as one of the productions of the non-terminal *s* of the grammar. It is less obvious how such non-terminal patterns should be dealt with in the unifier. To unify two such patterns, the intersection of two non-terminals should be computed, which reduces to the problem of computing the intersection of tree automata, for which there is no efficient algorithm (Comon et al. 2007).

Instead a conservative check is used at the time of unification. When unifying a non-terminal with another pattern, we attempt to unify the pattern with each production of the non-terminal, replacing any embedded non-terminal references with the pattern `any`. We require that at least one of the unifications succeeds. Because this is not a complete check for pattern intersection, we save the names of the non-terminals as extra information embedded in the constraint store until the entire generation process is complete. Then, once we generate a concrete term, we check to see if any of the non-terminals would have been violated (using a matching algorithm). This means that we can get failures at this stage of generation, but it tends not to happen very often for practical Redex models.[3]

[3]To be more precise, on the Redex benchmark (see section ???) such failures occur on all "delim-cont" models $2.9 \pm 1.1\%$ of the time, on all "poly-stlc" models $3.3 \pm 0.3\%$ of the time, on the "rvm-6" model $8.6 \pm 2.9\%$ of the time, and are not observed on the other models.

Bibliography

Franz Baader and Wayne Snyder. Unification Theory. *Handbook of Automated Reasoning* 1, pp. 445–532, 2001.

H. P. Barendregt. The Lambda Calculus: Its Syntax and Semantics. North Holland, 1984.

H. Comon, M. Dauchet, R. Gilleron, C. Loding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications. 2007. `http://www.grappa.univ-lille3.fr/tata`

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2010.

Matthias Felleisen and Robert Hieb. The Revised Report on the Syntactic Theories of Squential Control and State. *Theoretical Computer Science* 103(2), pp. 235–271, 1992.

Burke Fetscher. The Random Generation of Well-Typed Terms. Northwestern University, NU-EECS-14-05, 2014.

Robert Harper. Practical Foundations for Programming Languages. Cambridge University Press, 2012.

Joxan Jaffar, Michael J. Maher, Kim Marriott, and Peter J. Stuckey. The Semantics of Constraint Logic Programming. *Journal of Logic Programming* 37(1-3), pp. 1–46, 1998.

Casey Klein. Experience with Randomized Testing in Programming Language Metatheory. MS dissertation, Northwestern University, 2009.

Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. Run Your Research: On the Effectiveness of Lightweight Mechanization. In *Proc. ACM Symp. Principles of Programming Languages*, 2012.

Casey Klein, Robert Bruce Findler, and Matthew Flatt. The Racket virtual machine and randomized testing. *Higher-Order and Symbolic Computation*, 2013.