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

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 ??? (???)'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.

$$
\begin{array}{rl}
P & ::= (D \ldots) \\
D & ::= (r \ldots) \\
r & ::= ((d\ p) \leftarrow a \ldots) \\
a & ::= (d\ p) \mid \delta \\
d & ::= \mathit{Identifier}
\end{array}
\qquad
\begin{array}{rl}
C & ::= (\wedge\ (\wedge\ (x = p) \ldots) \\
 & \qquad (\wedge\ \delta \ldots)) \\
e & ::= (p = p) \\
\delta & ::= (\forall\ (x \ldots)\ (\vee\ (p \neq p) \ldots))
\end{array}
\qquad
\begin{array}{rl}
p & ::= (\mathsf{lst}\ p \ldots) \\
 & \mid m \\
 & \mid x \\
m & ::= \mathit{Constant} \\
x & ::= \mathit{Variable}
\end{array}
$$

Programs Formulas Patterns

Figure 1: The syntax of the derivation generator model.

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

Figure 2: Reduction rules describing generation of the complete tree of derivations.

The grammar in figure 1 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 1.1.

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 1 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 2 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 1.1. Section 1.2 describes how our solver handles equations and disequations. Section 1.3 discusses the heuristics in our implementation and section 1.4 describes how our implementation scales up to support features in Redex that are not covered in this model.

1.1. Compiling Metafunctions

The primary difference between a metafunction, as written in Redex, and a set of $((d\ p) \leftarrow a\ ...)$ clauses from figure 1 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:

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

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.

1.2. 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 1, 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 3 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 $\perp$, 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 (??? ???) 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 $\perp$.

Figure 4 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 $\perp$ then the disequation is inconsistent with the current constraint store and thus dissolve itself returns $\perp$. 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

$$\textsf{solve} : e\ C \rightarrow C \text{ or } \bot$$

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

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

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

$$\begin{aligned}
&\textsf{unify}[\![((p = p)\ e\ ...), (\wedge\ e_s\ ...)]\!] &&= \textsf{unify}[\![(e\ ...), (\wedge\ e_s\ ...)]\!] \\
&\textsf{unify}[\![(((\textsf{lst}\ p_1\ ..._1) = (\textsf{lst}\ p_2\ ..._1))\ e\ ...), (\wedge\ e_s\ ...)]\!] &&= \textsf{unify}[\![((p_1 = p_2)\ ...\ e\ ...), (\wedge\ e_s\ ...)]\!] \\
&\quad \text{where } |(p_{1 ...})| = |(p_2\ ...)| && \\
&\textsf{unify}[\![((x = p)\ e\ ...), (\wedge\ e_s\ ...)]\!] &&= \bot \\
&\quad \text{where } \textsf{occurs?}[\![x, p]\!], x \neq p && \\
&\textsf{unify}[\![((x = p)\ e\ ...), (\wedge\ e_s\ ...)]\!] &&= \textsf{unify}[\![(e\{x \rightarrow p\}\ ...), \\
& && \qquad (\wedge\ (x = p)\ e_s\{x \rightarrow p\}\ ...)]\!] \\
&\textsf{unify}[\![((p = x)\ e\ ...), (\wedge\ e_s\ ...)]\!] &&= \textsf{unify}[\![((x = p)\ e\ ...), (\wedge\ e_s\ ...)]\!] \\
&\textsf{unify}[\![(), (\wedge\ e\ ...)]\!] &&= (\wedge\ e\ ...) \\
&\textsf{unify}[\![(e\ ...), (\wedge\ e_s\ ...)]\!] &&= \bot
\end{aligned}$$

Figure 3: 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 $\bot$.

$$\mathsf{dissolve} : \delta\ C \rightarrow C \text{ or } \bot$$

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

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

$$\mathsf{disunify} : \delta \rightarrow \delta \text{ or } \top \text{ or } \bot$$

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

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

Figure 4: 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.

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

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

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

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

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

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

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

Figure 5: Metafunctions used to process disequational constaints.

This brings us to param-elim, in figure 5. 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 (??? ???).

Finally, we return to check, shown in figure 5, which is passed the updated disequations after a new equation has been added in solve (see figure 3). 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.

1.3. 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.

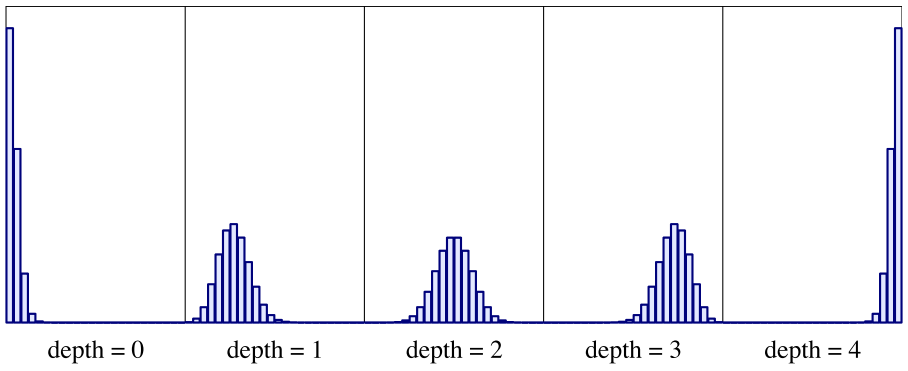

Figure 6: Density functions of the distributions used for the depth-dependent rule ordering, where the depth limit is 4 and there are 4 rules.

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

Figure 7: 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.

1.4. A Richer Pattern Language

The model we present in section 1 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 7. 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 1, 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 (??? ???). 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±1.1% of the time, on all "poly-stlc" models 3.3±0.3% of the time, on the "rvm-6" model 8.6±2.9% of the time, and are not observed on the other models.