NORTHWESTERN
UNIVERSITY

Electrical Engineering and Computer Science Department

Technical Report
Number: NU-EECS-14-05

June, 2014

The Random Generation of Well-Typed Terms

Burke Fetscher

Abstract

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 random generation often performs better on it than an older approach previously used by Redex.

Keywords: Automated Testing, Semantics

NORTHWESTERN UNIVERSITY

The Random Generation of Well-Typed Terms

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

MASTER OF SCIENCE

Field of Electrical Engineering and Computer Science

By

Burke Fetscher

EVANSTON, ILLINOIS

June 2014

ABSTRACT

The Random Generation of Well-Typed Terms

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 lighweight 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 this benchmark and show that our new approach to random generation does markedly better on it than an older approach included with Redex.

Note:

The ESOP 2015 paper *Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System* by Fetscher, Claessen, Pałka, Hughes, and Findler contains an updated presentation of much of the material in this dissertation, specifically a cleaner version of the derivation generator model, and a newer evaluation including an additional case study.

Table of Contents

CHAPTER 1

Introduction

Testing is universally considered an integral part of the software development process. It is less widely accepted as a useful tool for semantics engineering, where formal proofs are the tool of choice. Since its inception, PLT Redex [10] has challenged that assumption as a lightweight workbench for semantics modelling that utilizes testing, and specifically random testing, as an integral part of its toolbox. Redex has used a straightforward approach to random generation which has been effective as a basis for random testing. In this work, we enrich Redex's random testing capabilities by providing automatic support for generating terms that satisfy relations and functions defined in Redex. We then evaluate the potential of this approach to random testing by comparing it to several others.

Redex already provides significant support for random testing, inspired by the well-known Haskell testing library QuickCheck [5]. After writing a model of a semantics in Redex, users can define a property they wish to test and Redex is then able to randomly generate terms satisfying a grammar which are used as test cases to try to falsify the property. This approach, and variations thereof, has been used to successfully find counterexamples in many real-world semantics models. [18, 19, 20] However, it suffers from the deficiency that useful testable properties usually have a precondition that is stricter than the language grammar, so that the vast majority of randomly generated terms are not valid test cases.

To attempt to remedy this situation, we have extended Redex's "push-button" approach beyond language grammars with the capability to generate random terms from richer relations on terms

that are frequently defined as part of a semantic model. Type systems are a primary example. With the old method of random testing, Redex users would frequently define a property of *well-typed* terms, and then generate terms from a grammar, using the type system as a filter before testing the property. The new approach allows test cases to be automatically generated from the type system as the user has already written it down.

This approach to generating random terms is more complex and necessarily slower than using a grammar, so it is not immediately clear that it is more effective as a testing method. To evaluate its effectiveness, we first compared it to the old method on a benchmark of Redex models to which we have added bugs by hand. We find that the new generator does much better than this naive grammar-based method, finding bugs the old method does not and in much shorter times.

Because of the relative scarcity of test cases generated from a grammar, most Redex testing efforts incldue hand-written extensions to the automatic random generation capabilities. We have also compared the new random generator against a handcrafted test generator of this type, a pre-existing model of the Racket Virtual Machine [20] that already had many hours of development effort behind it. The handcrafted generator performs better than the new method, finding bugs faster and exposing one bug the new generator wasn't able to find.

This dissertation continues by giving a more in-depth tour of Redex and its test and random generation capabilities in Chapter 2. Chapter 3 then explains how the new method of random term generation works, and Chapter 4 details work to evaluate its effectiveness. Chapter 5 discusses related work and Chapter 6 concludes.

CHAPTER 2

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.

2.1. 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. A

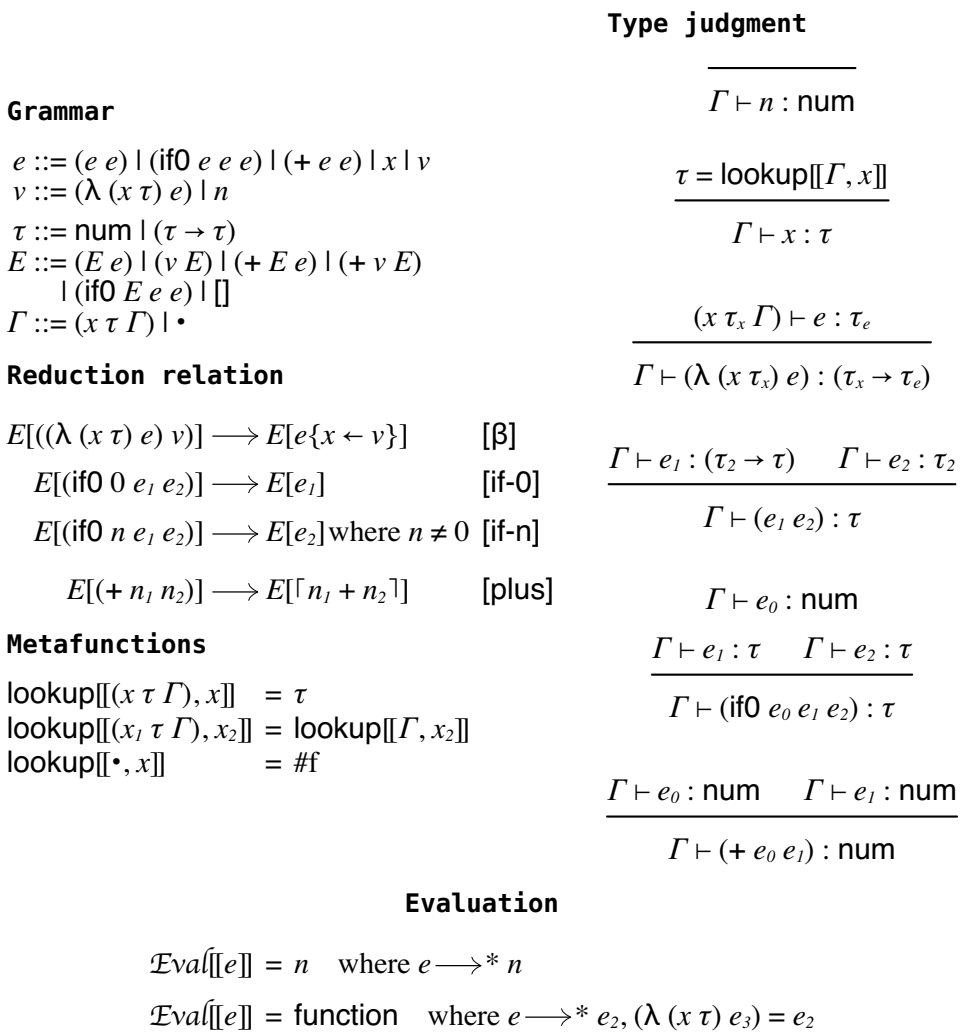


Figure 1: A reduction semantics and type system for the simply-typed lambda calculus.

thorough introduction to reduction semantics can be found in Felleisen et al. [10]; they were first introduced in Felleisen and Hieb [11].

Figure 1 shows a reduction semantics for the simply-typed lambda calculus. The lambda calculus [3] 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]

```
(define-language STLC
  (e ::= (e e) (if0 e e e) (+ e e) x v)
  (v ::= (λ (x τ) e) n)
  (n ::= number)
  (x ::= variable-not-otherwise-mentioned)
  (τ ::= num (τ → τ))
  (E ::= (E e) (v E) (+ E e) (+ v E)
         (if0 E e e) hole)
  (Γ ::= (x τ Γ) •))
```

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 nonterminal (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

[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.

and symbols that do not appear in productions in the grammar (λ is excluded, for example). Once a 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)
        (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.

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

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)
```

[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.

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 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 [14] 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

[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.

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))
   ----------------------
   (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 [14] 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

[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.

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
```

2.2. 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 [18] 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)
'((+ (U (if0 0 (if0 L V eo) 2)) Y) 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:
fp
```

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)))
```

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.

```
redex-check: no counterexamples in 1000 attempts
'(("if-0" . 0) ("if-n" . 0) ("plus" . 3) ("β" . 0))
```

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" 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 $(\lambda\ (x\ \tau)\ n)$). 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 [18], the Racket Virtual Machine [20], and various language models drawn from the International Conference on Functional Programming [19]. 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 3

Random Term Generation

This chapter explains the method used to generate terms satisfying judgment forms and metafunctions. To introduce the approach, it begins by working through an example generation for a well-typed term. The generation method is then explained in general, starting with a subset of Redex's pattern and term languages, which is used as a basis for describing judgment-form based generation. The generation method works by searching for derivations that satisfy the relevant relation definitions, using a constraint solver to maintain consistency during the search.

Following the explanation of the basic generation method, metafunction generation is introduced along with some necessary extensions to the constraint solver. Then the constraint solver is explained in some detail. Finally the methods used to handle Redex's full pattern language are discussed.

3.1. An Example Derivation

The judgment-form based random generator uses the strategy of attempting to generate a random derivation satisfying the judgment. To motivate how it works, we will work through the

$$\frac{}{\Gamma \vdash n : \mathsf{num}} \qquad \frac{\tau = \mathsf{lookup}[[\Gamma, x]]}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e_1 : (\tau_2 \to \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1\ e_2) : \tau}$$

$$\frac{\Gamma \vdash e_0 : \mathsf{num} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (\mathsf{if0}\ e_0\ e_1\ e_2) : \tau} \qquad \frac{(x\ \tau_x\ \Gamma) \vdash e : \tau_e}{\Gamma \vdash (\lambda\ (x\ \tau_x)\ e) : (\tau_x \to \tau_e)} \qquad \frac{\Gamma \vdash e_0 : \mathsf{num} \quad \Gamma \vdash e_1 : \mathsf{num}}{\Gamma \vdash (+\ e_0\ e_1) : \mathsf{num}}$$

Figure 3: Typing judgment for the simply-typed lambda calculus

generation of an example derivation for the type system shown in figure 3. We can begin with a schema for how we would like the resulting judgment to look. We would like to find a derivation for some expression e^0 with some type τ^0 in the empty environment:

$$\bullet \vdash e^0 \ : \ \tau^0$$

Where we have added superscripts to distinguish variables introduced in this step from those introduced later; since this is the initial step, we mark them with the index 0. The rule chosen in the initial generation step will be final rule of the derivation. The derivation will have to end with some rule, so we randomly choose one, suppose it is the abstraction rule. Choosing that rule will require us to specialize the values of e^0 and τ^0 in order to agree with the form of the rule's conclusion. Once we do so, we have a partial derivation that looks like:

$$\frac{(x^1 \ \tau_x^1 \ \bullet) \ \vdash \ e^1 \ : \ \tau^1}{\bullet \ \vdash \ (\lambda \ (x^1 \ \tau_x^1) \ e^1) \ : \ (\tau_x^1 \to \tau^1)}$$

Variables from this step are marked with a 1. This will work fine, so long as we can recursively generate a derivation for the premise we have added. We can randomly choose a rule again and try to do so.

If we next choose abstraction again, followed by function application, we arrive at the following partial derivation:

$$\frac{\dfrac{\dfrac{(x^2 \ \tau_x^2 \ (x^1 \ \tau_x^1 \ \bullet)) \ \vdash \ e_1^3 \ : \ (\tau_2^3 \to \tau^2) \quad (x^2 \ \tau_x^2 \ (x^1 \ \tau_x^1 \ \bullet)) \ \vdash \ e_2^3 \ : \ \tau_2^3}{(x^2 \ \tau_x^2 \ (x^1 \ \tau_x^1 \ \bullet)) \ \vdash \ (e_1^3 \ e_2^3) \ : \ \tau^2}}{(x^1 \ \tau_x^1 \ \bullet) \ \vdash \ (\lambda \ (x^2 \ \tau_x^2) \ (e_1^3 \ e_2^3)) \ : \ (\tau_x^2 \to \tau^2)}}{\bullet \ \vdash \ (\lambda \ (x^1 \ \tau_x^1) \ (\lambda \ (x^2 \ \tau_x^2) \ (e_1^3 \ e_2^3))) \ : \ (\tau_x^1 \to (\tau_x^2 \to \tau^2))}$$

Abstraction has two premises, so now there are two branches of the derivation that need to be filled in. We can work on the left side first. Suppose we make a random choice to use the variable rule there, and arrive at the following:

$$\begin{array}{lll}\mathsf{lookup}[[(x\ \tau\ \Gamma), x]] & = & \tau \\ \mathsf{lookup}[[(x_1\ \tau\ \Gamma), x_2]] & = & \mathsf{lookup}[[\Gamma, x_2]] \\ \mathsf{lookup}[[\bullet, x]] & = & \#\mathrm{f}\end{array} \qquad \frac{}{\mathsf{lookup}[[(x\ \tau\ \Gamma), x]] \ = \ \tau} \qquad \frac{}{\mathsf{lookup}[[\bullet, x]] \ = \ \#\mathrm{f}} \qquad \frac{x_1 \neq x_2 \quad \mathsf{lookup}[[\Gamma, x_2]] \ = \ \tau}{\mathsf{lookup}[[(x_1\ \tau_x\ \Gamma), x_2]] \ = \ \tau}$$

Figure 4: Lookup as a metafunction (left), and as a judgment form.

$$\dfrac{\dfrac{\dfrac{\mathsf{lookup}[[(x^2\ \tau_x^2\ (x^1\ \tau_x^1\ \bullet)), x^4]] \ = \ (\tau_2^3 \to \tau^2)}{(x^2\ \tau_x^2\ (x^1\ \tau_x^1\ \bullet)) \ \vdash \ x^4 \ : \ (\tau_2^3 \to \tau^2)} \qquad (x^2\ \tau_x^2\ (x^1\ \tau_x^1\ \bullet)) \ \vdash \ e_2^3 \ : \ \tau_2^3}{(x^2\ \tau_x^2\ (x^1\ \tau_x^1\ \bullet)) \ \vdash \ (x^4\ e_2^3) \ : \ \tau^2}}{(x^1\ \tau_x^1\ \bullet) \ \vdash \ (\lambda\ (x^2\ \tau_x^2)\ (x^4\ e_2^3)) \ : \ (\tau_x^2 \to \tau^2)}}{\bullet \ \vdash \ (\lambda\ (x^1\ \tau_x^1)\ (\lambda\ (x^2\ \tau_x^2)\ (x^4\ e_2^3))) \ : \ (\tau_x^1 \to (\tau_x^2 \to \tau^2))}$$

At this point it isn't obvious how to continue, because `lookup` is defined as a metafunction, and we are generating a derivation using a method based on judgment forms. To complete the derivation for `lookup`, we will treat it as a judgment form, except that we have to be careful to preserve its meaning, since judgment form cases don't apply in order and, in fact, the second case of `lookup` overlaps with the first. So that we can never apply the rule corresponding to the second case when we should be using the first, we will add a second premise to that rule stating that $x_1 \neq x_2$. The new version of `lookup` is shown in figure 4, alongside the original. If we now choose the `lookup` rule that recurs on the tail of the environment (corresponding to the second clause of the metafunction), the partial derivation looks like:

$$\dfrac{\dfrac{\dfrac{\dfrac{x^2 \ \neq \ x^4 \quad \mathsf{lookup}[[(x^1\ \tau_x^1\ \bullet), x^4]] \ = \ (\tau_2^3 \to \tau^2)}{\mathsf{lookup}[[(x^2\ \tau_x^2\ (x^1\ \tau_x^1\ \bullet)), x^4]] \ = \ (\tau_2^3 \to \tau^2)}}{(x^2\ \tau_x^2\ (x^1\ \tau_x^1\ \bullet)) \ \vdash \ x^4 \ : \ (\tau_2^3 \to \tau^2)} \qquad (x^2\ \tau_x^2\ (x^1\ \tau_x^1\ \bullet)) \ \vdash \ e_2^3 \ : \ \tau_2^3}{(x^2\ \tau_x^2\ (x^1\ \tau_x^1\ \bullet)) \ \vdash \ (x^4\ e_2^3) \ : \ \tau^2}}{(x^1\ \tau_x^1\ \bullet) \ \vdash \ (\lambda\ (x^2\ \tau_x^2)\ (x^4\ e_2^3)) \ : \ (\tau_x^2 \to \tau^2)}}{\bullet \ \vdash \ (\lambda\ (x^1\ \tau_x^1)\ (\lambda\ (x^2\ \tau_x^2)\ (x^4\ e_2^3))) \ : \ (\tau_x^1 \to (\tau_x^2 \to \tau^2))}$$

This branch of the derivation can be completed by choosing the rule corresponding to the first clause of `lookup` to get:

$$\dfrac{\dfrac{x^2 \neq x^1 \quad \dfrac{}{\mathsf{lookup}[[(x^1\ (\tau_2^3 \to \tau^2)\ \bullet), x^1]] = (\tau_2^3 \to \tau^2)}}{\mathsf{lookup}[[(x^2\ \tau_x^2\ (x^1\ (\tau_2^3 \to \tau^2)\ \bullet)), x^1]] = (\tau_2^3 \to \tau^2)}}{\dfrac{(x^2\ \tau_x^2\ (x^1\ (\tau_2^3 \to \tau^2)\ \bullet)) \vdash x^1 : (\tau_2^3 \to \tau^2) \qquad (x^2\ \tau_x^2\ (x^1\ (\tau_2^3 \to \tau^2)\ \bullet)) \vdash e_2^3 : \tau_2^3}{(x^2\ \tau_x^2\ (x^1\ (\tau_2^3 \to \tau^2)\ \bullet)) \vdash (x^1\ e_2^3) : \tau^2}}{\dfrac{(x^1\ (\tau_2^3 \to \tau^2)\ \bullet) \vdash (\lambda\ (x^2\ \tau_x^2)\ (x^1\ e_2^3)) : (\tau_x^2 \to \tau^2)}{\bullet \vdash (\lambda\ (x^1\ (\tau_2^3 \to \tau^2))\ (\lambda\ (x^2\ \tau_x^2)\ (x^1\ e_2^3))) : ((\tau_2^3 \to \tau^2) \to (\tau_x^2 \to \tau^2))}}$$

It is worth noting at this point that the form of the partial derivation may sometimes exclude rules from being chosen. For example, we couldn't satisfy the right branch of the derivation in the same way, since that would eventually mean that $\tau_2^3 = (\tau_2^3 \to \tau^2)$, leaving us with no finite value for τ_2^3. However, we can complete the right branch by again choosing (randomly) the variable rule, followed by the rule corresponding to `lookup`'s first clause, arriving at:

$$\dfrac{\dfrac{x^2 \neq x^1 \quad \dfrac{}{\mathsf{lookup}[[(x^1\ (\tau_x^2 \to \tau^2)\ \bullet), x^1]] = (\tau_x^2 \to \tau^2)}}{\mathsf{lookup}[[(x^2\ \tau_x^2\ (x^1\ (\tau_x^2 \to \tau^2)\ \bullet)), x^1]] = (\tau_x^2 \to \tau^2)} \quad \dfrac{}{\mathsf{lookup}[[(x^2\ \tau_x^2\ (x^1\ (\tau_x^2 \to \tau^2)\ \bullet)), x^2]] = \tau_x^2}}{\dfrac{(x^2\ \tau_x^2\ (x^1\ (\tau_x^2 \to \tau^2)\ \bullet)) \vdash x^1 : (\tau_x^2 \to \tau^2) \qquad (x^2\ \tau_x^2\ (x^1\ (\tau_x^2 \to \tau^2)\ \bullet)) \vdash x^2 : \tau_x^2}{(x^2\ \tau_x^2\ (x^1\ (\tau_x^2 \to \tau^2)\ \bullet)) \vdash (x^1\ x^2) : \tau^2}}{\dfrac{(x^1\ (\tau_x^2 \to \tau^2)\ \bullet) \vdash (\lambda\ (x^2\ \tau_x^2)\ (x^1\ x^2)) : (\tau_x^2 \to \tau^2)}{\bullet \vdash (\lambda\ (x^1\ (\tau_x^2 \to \tau^2))\ (\lambda\ (x^2\ \tau_x^2)\ (x^1\ x^2))) : ((\tau_x^2 \to \tau^2) \to (\tau_x^2 \to \tau^2))}}$$

At this point we have a complete derivation for a pattern of non-terminals that is valid for any term that matches that pattern as long as the new premise that $x^2 \neq x^1$ is also satisfied. Thus we can simply pick appropriate random values for x^1 and all other non-terminals in the pattern to get a random term that satisfies the typing judgment. An example would be:

$$\bullet \vdash (\lambda\ (\mathsf{f}\ (\mathsf{num} \to \mathsf{num}))\ (\lambda\ (\mathsf{a}\ \mathsf{num})\ (\mathsf{f}\ \mathsf{a}))) : ((\mathsf{num} \to \mathsf{num}) \to (\mathsf{num} \to \mathsf{num}))$$

and the constraint that $\mathtt{f} \neq \mathtt{a}$ is satisfied. We note however, the importance of this constraint, since a term that does not satisfy it, such as $(\lambda\ (\mathsf{f}\ (\mathsf{num} \to \mathsf{num}))\ (\lambda\ (\mathsf{f}\ \mathsf{num})\ (\mathsf{f}\ \mathsf{f})))$, is not well-typed.

In the remainder of this chapter, the approach used in this example is generalized to all Redex judgment forms and metafunctions.

p ::= (list p ...)	t ::= (t ...)	a ::= *Racket Constant*
| x	| (f t ...)	x ::= *Variable*
| a	| x	f ::= *Metafunction*
| b	| a	b ::= *Built-in Pattern*

Figure 5: Simplified grammar for Redex patterns (p) and terms (t).

3.2. Patterns and Terms in Redex

Redex handles two s-expression-based grammars internally: patterns and terms. Simplified forms of both are shown in figure 5. Terms t are essentially s-expressions built from Racket constants a, except that Redex provides a term environment with bindings for term variables[1] x and metafunctions f. When a metafunction is a applied in a term, the result will be a term, and the result term is inserted in place of the application. Term variables are simply bound to other terms and are replaced by their values. They are bound at the successful match of a pattern variable.

Patterns p are used to match against and decompose terms. They are composed of literals a which match themselves only, and built in patterns b which match some set of literals — `number`, which matches any Racket number, is one example. Pattern variables `x` match against a term and bind the variable to the term. Finally, lists of patterns may be matched against lists of terms.

The term generator actually operates on patterns and produces a pattern as an intermediate result. The conversion of the resulting pattern to a term is straightforward. As a first step in generation, then, terms in judgment form and metafunction definitions are converted into corresponding patterns (as described in more detail below).

3.3. Judgment Form Generation

A judgment form in Redex is defined in the following manner, as a set of inference rules:

[1]Variables are just Racket symbols that are bound or in binding positions in Redex's term context. In pattern they are in a binding position, and are bound when they appear in a term.

$$J ::= id$$
$$s ::= p \mid t$$

$$\frac{(J_1^1\ s_1^1\ \ldots)\ \ldots\ (J_m^1\ s_m^1\ \ldots)}{(J_c\ s_c^1\ \ldots\ s_c^k)}\ \ldots\ \frac{(J_1^n\ s_1^n\ \ldots)\ \ldots\ (J_m^n\ s_m^n\ \ldots)}{(J_c\ s_c^1\ \ldots\ s_c^k)}$$

Where the J non-terminals indicate judgment form ids, and a single judgment form is defined by a set of rules with matching ids in the conclusion. Note that that the number of patterns and terms in the conclusion of a single judgment form must be a constant k. The premise of a single rule (above the line) consists of the conjunction of some further set of judgments. To derive the conclusion of a judgment, there must exist derivations of all of its premises; the complete derivation of a judgment is the tree generated by satisfying all such recursive derivations. A judgment form J then inductively defines a relation over a k-tuple of terms, such that the k-tuple $\langle t_1, \ldots, t_k \rangle$ is in the relation if there exists a complete derivation of $(J\ t_1\ \ldots\ t_k)$ using the available inference rules.

In Redex, judgment forms are required to have a specified mode determining which positions in the judgment definition are inputs, and which are outputs. (These are also sometimes referred to as positive and negative positions.) In the conclusion of a judgment, input mode positions are patterns, which deconstruct input terms, and in the premises, input positions are considered terms, which are used as inputs to recursive attempts to satisfy the judgment. Output positions in premises are patterns which deconstruct the results of recursive calls, and in the conclusion output positions are terms which are the result of trying to satisfy the judgment for some set of inputs. Pattern positions may bind parts of a successful match for use in building terms in the term positions. Thus a judgment may be executed as a function by providing input terms, the result of which will be some (possibly empty) set of term tuples corresponding to possible values of the output positions of the judgment.

For random generation of terms satisfying a judgment, however, it isn't practical to maintain the distinction between modes of different judgment positions, because it is very difficult to successfully pick a set of input terms that will satisfy a judgment. Instead we choose to attempt to construct a random derivation tree, maintaining the invariant that a partial derivation is valid as we do so. Since this precludes choosing values for term position, all positions of the judgment must be treated as patterns. Thus the judgment is pre-processed by traversing all pattern positions in the appropriate order to extract binding identifiers, which are used to rewrite terms in term positions into patterns. Binding identifiers then create constraints between the patterns in the rule, as the same identifier may appear in multiple patterns. Metafunction applications are also transformed during this this step, as is explained in the next section on metafunction generation.

To try to generate terms satisfying a given judgment J, we can attempt to construct some random derivation that ends with of one the rules defining J. A randomly chosen rule will have the form:

$$\frac{(J_1\ p_1\ ...)\ \ ...\ \ (J_m\ p_m\ \ ...)}{(J\ p_c\ ...)}$$

(Where the p's are meant to reflect that all positions have now been rewritten as patterns.) If this rule is chosen, in order to complete the derivation, m sub-derivations must be generated as well, one for each judgment $(J_k\ p_k\ ...)$ in $k = 1...m$. Generation thus proceeds recursively, generating goals of the form $(J_g\ p_g\ ...)$ which are to be filled in with subderivations. In general, a rule with a conclusion $(J_c\ p_c\ ...)$ can be used to attempt to generate a derivation for the goal $(J_g\ p_g\ ...)$ if rule defines the correct judgment, i.e. $J_c = J_g$, and the set of equations $\{p_c = p_g, ...\}$ has solutions. Thus a derivation will generate a set of equational constraints, which are solved by successively unifying the patterns in each equation. The result of unification is a substitution for the pattern variables that satisfies the constraints, or failure, if such a substitution does not exist. (Unification and

$$S ::= (P \vdash G \parallel C)$$

$$P ::= (R \ ...) \qquad G ::= (g \ ...)$$
$$R ::= (L \leftarrow L \ ...) \qquad g ::= c \mid L$$
$$L ::= (J \ p \ ...) \qquad c ::= (p = p)$$

$$(P \vdash (c_g \ g \ ...) \parallel C_0) \longrightarrow (P \vdash (g \ ...) \parallel C) \quad \text{[new constraint]}$$
$$\text{where } C = \text{add-constraint}[[C_0, c_g]]$$

$$(P \vdash ((J \ p_g \ ...) \ g \ ...) \parallel C) \longrightarrow (P \vdash ((p_f = p_g) \ ... \ L_f \ ... \ g \ ...) \parallel C) \quad \text{[reduce]}$$
$$\text{where } ((J \ p_r \ ...) \leftarrow L_r \ ...) = \text{select}[[J, P]],$$
$$((J \ p_f \ ...) \leftarrow L_f \ ...) = \text{freshen}[[((J \ p_r \ ...) \leftarrow L_r \ ...)]]$$

$$(P \vdash (c_g \ g \ ...) \parallel C_0) \longrightarrow (P \vdash () \parallel \bot) \quad \text{[invalid constraint]}$$
$$\text{where } \bot = \text{add-constraint}[[C_0, c_g]]$$

$$(P \vdash ((J \ p_g \ ...) \ g \ ...) \parallel C) \longrightarrow (P \vdash () \parallel \bot) \quad \text{[invalid literal]}$$
$$\text{where } \bot = \text{select}[[J, P]]$$

Figure 6: Derivation generation

disunification over Redex patterns are addressed in more detail Section 3.5.) The final substitution can be applied to the original goal to extract terms satisfying the judgment.

The derivation procedure is presented as a set of reduction rules in figure 6. The rules presented here are based on the operational semantics for constraint logic programming [16], used for their clarity and extensibility with respect to the constraint domain, as it will be necessary to add some new constraints to deal with metafunctions. The rules shown correspond exactly to those in Jaffar et al. [16], meaning the derivation generator is actually a random constraint logic programming system.

The rules operate on a program state S, which consists of the program P, the current goal G, and the current constraint store C. A "program" corresponds to judgment form definitions in Redex, and consists of a set of inference rules $(L \leftarrow L \ ...)$, written here such that the conclusion is left of the arrow and the premises are on the right. The current goal is a list of literals L, which correspond to

subderivations yet to be completed, and constraints c, which are just equations between terms. The constraint store is either C, which represents a consistent set of constraints, or $\perp$, which represents inconsistent constraints and indicates a failed derivation. (For simplicity the constraint store is kept opaque for the moment.)

The rules process the current goal and modify the constraint store until the goal is empty, at which point the derivation process is finished. When a constraint is the first element in the goal, it is checked for consistency with the procedure `add-constraint`, which returns an updated current constraint store on success or $\perp$ on failure (rules `new constraint` and `invalid constraint`).

When a literal $(J\ p_g\ ...)$ is the first element of the goal, the procedure `select` is used to choose a rule from the program that can be used to satisfy the indicated subderivation. The rule must be from the correct judgment (the judgment id of its conclusion must match that of the literal), and if such a rule cannot be found `select` may fail (rule `invalid literal`). Otherwise, the `reduce` rule applies and the rule is freshened to yield an instance $((J\ p_f\ ...) \leftarrow L_f\ ...)$ of the rule with uniquely named variables. Then constraints $p_g = p_f ...$ equating the patterns in the goal literal and the conclusion of the rule are added to the current goal, along with the premises $L_f ...$ of the chosen rule, which all must now be satisfied to complete the derivation.

The specification of the `select` function is especially important from the standpoint of random generation. The rules for derivation generation are deterministic aside from the behavior of `select`, which may affect the form of a derivation by varying the rule used to attempt to satisfy a literal. To generate a *random* derivation, `select` simply chooses randomly among the set of valid candidate rules for a given literal goal. However, this behavior can easily lead to non-terminating or inconveniently large derivations, since the `reduce` rule may expand the size of the goal. To account for this, once a certain depth bound is reached, rules are selected according to the number

of premises they have, from least to greatest. This makes it much more likely for a derivation to terminate finitely.

Finally, the model shown in figure 6 doesn't address the search behavior of the implementation. Specifically, when an attempted derivation results in failure, the generator *backtracks* to the state before most recent application of the `reduce` rule and tries again, with the constraint that `select` is no longer allowed to choose the rule that led to the failed derivation. This introduces the possibility of the search getting stuck in an arbitrarily long cycle, which is avoided by introducing a bound on the number of times such backtracking can occur in a single attempt to generate a derivation.

3.4. Metafunction Generation

In this section the requirements of adding support for metafunctions to term generation are considered. Aside from generating inputs and outputs of metafunctions directly, we have to handle the fact that metafunction application may be embedded inside any term, specifically term positions in judgment forms and metafunctions themselves. This is dealt with during the preprocessing phase that transforms terms into patterns by lifting out all metafunction applications and providing variable bindings for the result of the application. The applications are then added as premises of the rules used by the term generator. Exactly what it means for a metafunction application to be the premise of a rule will become clear as metafunction compilation and generation is explained.

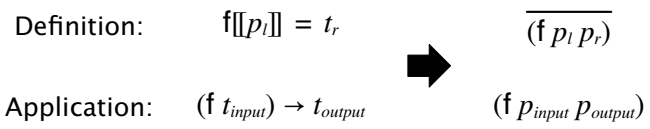
Metafunctions are defined in Redex as a sequence of cases, each of which has some sequence of left-hand-side patterns and a right-hand-side, or result, term:

$$\begin{array}{l} \mathsf{f}[\![p_1]\!] = t_1 \quad p ::= \mathit{pattern} \\ \quad\vdots \qquad\qquad\quad t ::= \mathit{term} \\ \mathsf{f}[\![p_n]\!] = t_n \end{array}$$

Where the p's are the left-hand-side patterns, and the t's are the result terms for the n clauses. Metafunctions are applied to terms, and metafunction application attempts to match the argument

term against the left-hand-side patterns in order from *1* to *n*. The result is the term corresponding to the first successful match, i.e. if the pattern p_k from clause k matches the input term, and none of the patterns p_1 ... p_{k-1} does, then the term t_k is the result. The pattern may bind parts of the match for use in constructing the result term.

To handle metafunctions in the derivation generation framework discussed in the previous section, the strategy of treating them as relations is adopted. First metafunctions are preprocessed in the same way as judgment forms, transforming terms into patterns and lifting out metafunction applications. Then, for a clause with left-hand side p_l and right-hand side p_r, a rule with the conclusion $(f\ p_l\ p_r)$ is added, where f is the metafunction name. For a metafunction application of $(f\ p_{input})$ with result p_{output}, a premise of the form $(f\ p_{input}\ p_{output})$ is added, and p_{output} is inserted at the location of the application:



The lifting of applications to premises occurs in both metafunction and judgment-form compilation, and transforms recursive metafunctions into recursive judgments. For example, if the term t_r in the illustration above contained a call to f, that call would become a premise of the resulting judgment, and its position in the pattern p_r would be replaced by the variable in the output position of the premise.

This translation accomplishes the goal of producing judgments as inputs for the generation scheme, but it doesn't preserve the semantics of metafunctions. Treating a metafunction definition as a relation ignores the ordering of the metafunction clauses. For a metafunction f with left-hand-side patterns $p_1 ... p_n$, if the generator attempts to satisfy a goal of the form $(f\ p_g)$ with clause k, a constraint of the form $p_k = p_g$ will be added. But it is possible that p_g is eventually instantiated as

some term that would have matched some previous pattern p_j, $1 \leq j < k$. In this case, an application of f to the term in question *should* have used clause j, but the generator has instead generated an application that used clause k. To avoid this situation, constraints that exclude the possibility of matching clauses *1* through k - *1* must be added; to generate an application that uses clause k the necessary condition becomes $p_k = p_g \wedge p_1 \neq p_g \wedge ... \wedge p_{k\text{-}1} \neq p_g$.

This seems sufficient at first, but further thought shows this constraint is not quite correct. Consider the following definition of the metafunction g:

$$\begin{aligned} g[\![(p_1\ p_2)]\!] &= 2 \\ g[\![p]\!] &= 1 \end{aligned}$$

Where in this context we can consider p to be pattern variable that will match any pattern, equivalent to `any` in Redex. Suppose when generating an application $(g\ p_{in})$ of this metafunction the second clause is chosen. This will generate the constraint $p_{in} = p \wedge p_{in} \neq (p_1\ p_2)$. (The fact that variables aside from p_{in} will be freshened is elided.) Now suppose that later on in the generation process, the constraint $p_{in} = (p_3\ p_4)$ is added, so the relevant part of the constraint store will be equivalent (after a bit of simplification) to:

$$p_{in} = (p_3\ p_4) \wedge p_1 \neq p_3 \wedge p_2 \neq p_4$$

The problem at this point is that it is possible to satisfy these constraints simply by choosing p_3 to be anything other than p_1, or p_4 anything other than p_2, but p_{in} will still be a two element list and thus would match the first clause of the metafunction.

The constraint excluding the first clause must be strong enough to disallow *any* two element list, which can be satisfied by requiring that:

$$\forall\, p_1\ \forall\, p_2\ p_{in} \neq (p_1\ p_2)$$

This suggests the general recipe for transforming metafunctions into judgments suitable for use in the derivation generator. Each clause is transformed into a rule as described above, with the addition of premises that are primitive constraints excluding the previous rules. For example, if clause k is being processed, the constraints will be of the form $\forall\, x \ldots\ p_k \neq p_i$, for $i = 1 \ldots k\text{-}1$, where p_k is the left hand side of clause k. There will be one constraint for each previous clause where the disallowed pattern p_i is the left hand side pattern of clause i, and all of the variables in p_i must be universally quantified, i.e. for constraint i, $\mathit{Variables}(p_i) = \{x \ldots\}$.

The derivation generation framework of figure 6 can easily be modified to handle the addition of the new constraints, the c non-terminal is simply extended with a single new production to be:

$$\begin{aligned} c ::= &\ (p = p) \\ &\mid (\forall\ (x \ldots)\ p \neq p) \end{aligned}$$

Disequational constraints in a judgment resulting from a metafunction transformation are added to the goal by the `reduce` rule and are handled in the same way as the equational constraints by the `new constraint` rule, provided that the constraint solver, no longer a simple unification algorithm, can deal with both types of constraints. The constraint solver and its extension to deal with disequations is discussed in the next section.

3.5. Equational and Disequational Constraints

This section presents a model of the constraint solver that operates on a simplified language. First, the operation of the solver on exclusively equational constraints, where it performs straightforward syntatic unification, is presented. Then the extension of the unifier to handle the form of disequational constraints introduced in the previous section is discussed. Finally issues specific to Redex's full pattern language are addressed.

$$
\begin{array}{ll}
P ::= (c\ ...) & c ::= eq \mid dq \\
S ::= ((x = t)\ ...) & eq ::= (s = t) \\
D ::= ((\forall\ (x\ ...)\ (s \neq t))\ ...) & dq ::= (\forall\ (x\ ...)\ (s \neq t)) \\
 & s, t ::= (\mathsf{f}\ t\ ...) \mid x
\end{array}
$$

Figure 7: Grammar for the constraint solver model.

The grammar for the constraint solver model is shown in figure 7. The model operates on the simplified term language of the *t* non-terminal, which has only two productions, one for `f`, a single multi-arity term constructor, and one for variables *x*. This corresponds closely to the Redex pattern language, which also has one multi-arity constructor, `list`. For now other complexities of the pattern language are ignored, as they don't directly impact the operation of the constraint solver. A problem *P* is a list of constraints *c*, which can be equations between terms *eq* or disequations *dq* (where some variables in the disequations are considered to be universally quantified). Given a problem, the solver constructs (and maintains, as the problem is extended) a substitution *S* that validates the equations and a set of simplified disequations *D*, such that *S* also validates *D*, and if *D* is valid, so are all the original disequations in *P*. The substitution[2] is written as a set of equations *x*=*t* between variables and terms, with the understanding that it can be *applied* to a term by, for each equation, finding each occurrence of *x* in the term and replacing it with *t*. A substitution validates an equation if both sides of the equation are syntactically identical after the substitution is applied.

3.5.1. Syntactic Unification

The portion of the solver that deals with equational constraints performs straightforward syntactic unification of patterns. The algorithm is well known; Baader and Snyder [2] provide a particularly

[2]To be more precise, *S* is actually the equational representation of some substitution γ, where γ is defined by its operation on terms. We will refer to the two interchangeably unless it is necessary to differentiate between them.

$$
\begin{array}{ll}
\mathsf{U} : P\ ,\ S\ ,\ D \rightarrow \textit{(S : D)}\ \text{or}\ \bot & \\
\mathsf{U}\llbracket((t = t)\ c\ ...), S, D\rrbracket & = \mathsf{U}\llbracket(c\ ...), S, D\rrbracket \\
\mathsf{U}\llbracket(((\mathsf{f}\ s\ ..._I) = (\mathsf{f}\ t\ ..._I))\ c\ ...), S, D\rrbracket & = \mathsf{U}\llbracket((s = t)\ ...\ c\ ...), S, D\rrbracket \quad \text{where } |(s\ ...)| = |(t\ ...)| \\
\mathsf{U}\llbracket(((\mathsf{f}\ s\ ..._{I\_1}) = (\mathsf{f}\ t\ ..._{I\_1}))\ c\ ...), S, D\rrbracket & = \bot \\
\mathsf{U}\llbracket((x = t)\ c\ ...), S, D\rrbracket & = \bot \quad \text{where } \mathsf{occurs?}[[x, t]], x \neq t \\
\mathsf{U}\llbracket((x = t)\ c\ ...), (c_s\ ...), (dq\ ...)\rrbracket & = \mathsf{U}\llbracket(c\{x \rightarrow t\}\ ...), ((x = t)\ c_s\{x \rightarrow t\}\ ...), (dq\{x \rightarrow t\}\ ...)\rrbracket \\
\mathsf{U}\llbracket((t = x)\ c\ ...), S, D\rrbracket & = \mathsf{U}\llbracket((x = t)\ c\ ...), S, D\rrbracket \\
\mathsf{U}\llbracket((), S, D\rrbracket & = (S : D)
\end{array}
$$

Figure 8: The metafunction `U` performs unification over the language of figure 7 . (Cases apply in order.)

good survey of theory in this area, including a similar presentation of syntactic unification that goes into greater detail.

The metafuction that performs unfication, U, is shown in figure 8. It operates on a problem *P*, a current substitution (or solution) *S*, and a current set of disequational constraints *D*, which is ignored by U. (Except in one case that will be addressed along with the disequational portion of the solver.) The result of U is either the the pair *(S : D)* of the substitution *S* and the disequations *D* that validate the entire problem, or $\bot$, if the problem is inconsistent.

The cases of U apply in order and dispatch on the first equation in *P*. In the first case, the equation is between two identical terms, and the equation is dropped before recurring on the rest of *P*. In the second case, the equation is between two terms applying the function constructor `f` to the same number of arguments; here the arguments in each position are equated and added to *P* before recurring. In the third case, two terms are contructed with `f` but have different numbers of arguents, in this case U fails and returns $\bot$ since it is impossible to make the terms equal. The fourth case equates a variable *x* with a term that contains the variable, which again leads to failure. (The metafunction `occurs?` takes a variable and a term, and returns true if the term

contains the variable, false otherwise.) The fifth case equates a variable x and a term t, where it is known (because the fourth case has already been attempted) that x does not occur in t. In this case t is substituted for x in the remaining equations of the problem c..., the equations of the current substitution c_s..., and the current disequations dq..., after which the equation x=t itself is added to the current substitution before recurring. The second to last case of U simply commutes an equation with a variable on the right hand side before recurring, after which the equation in question will be handled by one of the previous two cases. The final case returns S and D as the solution if the problem is empty.

To make a precise statement about the correctness of U, a few definitions are necessary. In the following, for a term t and substitution σ, σt is written to mean the application of σ to t.

Given two substitutions σ = *((x_σ = t_σ) ...)* and θ = *((x_θ = t_θ) ...)*, their *composition*, written $\theta\sigma$, is $\theta\sigma$ = *((x_σ = θt_σ) ... (x_θ = t_θ) ...)*, where trivial bindings of the form *(x = x)* are removed and when there is a duplicate binding *(x_σ = θt_σ)* and *(x_θ = t_θ)* where $x_\sigma = x_\theta$, then *(x_θ = t_θ)* is removed.

Two substitutions σ and θ are *equal*, $\sigma = \theta$, if for any variable x, $\sigma x = \theta x$. A substitution σ is *more general* than a substitution θ, written $\sigma \leq \theta$, if there exists some substitution γ such that $\theta = \gamma\sigma$.

A substitution σ is called the *most general unifier* of two terms s and t if $\sigma s = \sigma t$ and for any substitution γ such that $\gamma s = \gamma t$, $\sigma \leq \gamma$. Similarly, σ is a unifier for a unification problem P = *((s = t) ...)* if for every equation $s = t$ in P, $\sigma s = \sigma t$. It is a most general unifier for P if for every γ that is a unifier of P, $\sigma \leq \gamma$.

Finally, we can state that U is correct (again, ignoring the D part of the result for now):

> **Theorem 1** For any problem P, $\mathsf{U}(\!|P, (), ()|\!)$ terminates with $\perp$ if the equations in P have no unifier. Otherwise, it terminates with $(S_{mgu} : D)$ where S_{mgu} is a most general unifier for P.

Proofs of this proposition can be found in many references on unification, for example Baader and Snyder [2].

The version of U shown in figure 8 corresponds fairly closely with the implementation in Redex, except that the current substitution is represented as a hash table and the function recurs on the structure of input terms instead of using the current problem as a stack (as in the decomposition rule). As shown here, U has exponential complexity in both time and space. The space complexity arises because the substitution may have many identical terms, so by using a hash table it may be represented as a DAG (instead of a tree) with sharing between identical terms and linear space. However the worst-case running time is still exponential. Interestingly, this is still the most common implementation of unification because in practice the exponential running time never occurs, and in fact it is usually faster than algorithms with polynomial or near-linear worst-case complexity. [15]

3.5.2. Solving Disequational Constraints

This section extends the constraint solver of the previous section to process disequational constraints of the form $(\forall\ (x\ ...)\ (s \neq t))$ as well as the equational constraints already discussed. To handle disequations, U is extended with four new clauses. The new function is called DU, as this form of constraint solving is sometimes referred to as disunification [8]. The new clauses are shown in figure 9 along with the auxiliary metafunction `param-elim`. We now provide an informal explanation of DU's operation. A formal justification can be found in Appendix A.

$$\mathsf{DU} : P\ ,\ S\ ,\ D \rightarrow (S : D) \text{ or } \bot$$
$$\mathsf{DU}\,⦇(((\forall\ (x\ ...)\ (s \neq t))\ c\ ...), S, (dq\ ...)⦈\ = \bot$$
$$\text{where } (() : ()) = \mathsf{param\text{-}elim}[\![\mathsf{U}\,⦇((s = t)), (), ()⦈\ , (x\ ...)]\!]$$
$$\mathsf{DU}\,⦇(((\forall\ (x\ ...)\ (s \neq t))\ c\ ...), S, (dq\ ...)⦈\ =\ \mathsf{DU}\,⦇(c\ ...), S, (dq\ ...)⦈$$
$$\text{where } \bot = \mathsf{param\text{-}elim}[\![\mathsf{U}\,⦇((s = t)), (), ()⦈\ , (x\ ...)]\!]$$
$$\mathsf{DU}\,⦇(((\forall\ (x\ ...)\ (s \neq t))\ c\ ...), S, (dq\ ...)⦈\ =$$
$$\mathsf{DU}\,⦇(c\ ...), S, ((\forall\ (x\ ...)\ ((\mathsf{f}\ x_s\ ...) \neq (\mathsf{f}\ t_s\ ...)))\ dq\ ...)⦈$$
$$\text{where } (((x_s = t_s)\ ...) : ()) = \mathsf{param\text{-}elim}[\![\mathsf{U}\,⦇((s = t)), (), ()⦈\ , (x\ ...)]\!]$$
$$\mathsf{DU}\,⦇(c\ ...), S, (c_1\ ...\ (\forall\ (x_a\ ...)\ ((\mathsf{f}\ (\mathsf{f}\ s\ ...)\ ...)\ \neq (\mathsf{f}\ t\ ...)))\ c_2\ ...)⦈\ =$$
$$\mathsf{DU}\,⦇((\forall\ (x_a\ ...)\ ((\mathsf{f}\ (\mathsf{f}\ s\ ...)\ ...) \neq (\mathsf{f}\ t\ ...)))\ c\ ...), S, (c_1\ ...\ c_2\ ...)⦈$$

$$\mathsf{param\text{-}elim} : (S : ()) \text{ or } \bot\ ,\ (x\ ...) \rightarrow (S : ()) \text{ or } \bot$$
$$\mathsf{param\text{-}elim}[\![(((x_0 = t_0)\ ...\ (x = t)\ (x_1 = t_1)\ ...) : ()), (x_2\ ...\ x\ x_3\ ...)]\!] =$$
$$\mathsf{param\text{-}elim}[\![(((x_0 = t_0)\ ...\ (x_1 = t_1)\ ...) : ()), (x_2\ ...\ x\ x_3\ ...)]\!]$$
$$\mathsf{param\text{-}elim}[\![(((x_0 = t_0)\ ...\ (x_1 = x)\ c_2\ ...) : ()), (x_2\ ...\ x\ x_3\ ...)]\!] =$$
$$\mathsf{param\text{-}elim}[\![(((x_0 = t_0)\ ...\ (x_1 = x_2)\ ...\ c_3\ ...) : ()), (x_2\ ...\ x\ x_3\ ...)]\!]$$
$$\text{where } x \notin (t_0\ ...), ((x_1 = x_2)\ ...\ c_3\ ...) = \mathsf{elim\text{-}x}[\![x, (x_1 = x), c_2, ...]\!]$$
$$\mathsf{param\text{-}elim}[\![\bot, (x\ ...)]\!]\ =\ \bot$$
$$\mathsf{param\text{-}elim}[\![(S : ()), (x\ ...)]\!]\ =\ (S : ())$$

Figure 9: Extensions to figure 8 to handle disequational constraints. DU extends U with four new clauses.

The first three clauses of DU all address the situation where the first constraint in the problem P is a disequation of the form $(\forall\ (x\ ...)\ \ (s \neq t))$. Actually in all three cases, the metafunction U (recall that U is the portion of the solver that applies exclusively to equations) is called with a problem containing the single equation $(s = t)$ and an empty substitution. The result of this call is passed to the metafunction `param-elim` along with a list of the parameters, which is where special handling of the universally quantified variables takes place. (Borrowing the terminology of Comon and Lescanne [8], the universally quantified variables are referred to as "parameters".) It is the result of this process that determines which of the first three cases of DU applies. Of course, in the Redex implementation, the calls to U and `param-elim` occur only once.

The call to U will return either $\perp$ or *(S : ()))*. If the result is $\perp$, then `param-elim` does nothing and DU simply drops the constraint in question and recurs. (This is the second clause of DU.) The reasoning here is that it is impossible to unify the terms, so the disequation will always be satisfied.

If U returns a value of the form *(S : ()))*, then S is a most general unifier for the equation in question, so for any substitution γ such that $\gamma s = \gamma t$, $S \leq \gamma$. Thus S is used to create a new constraint excluding all such γ. In the absence of parameters, for S = *((x = t_x) ...)*, this would involve simply adding a constraint of the form *(x $\neq$ t_x)* $\vee$ ..., since validating any one of the disequations excludes S (and by doing so excludes all γ where $\gamma \leq S$). If S contains any parameters not underneath a function constructor, they are eliminated by `param-elim`, the intuition being that it is impossible to satisfy a disequation of the form *(∀ (x) (x $\neq$ t))* since x cannot be chosen to be a term other than t.

The auxiliary metafunction `param-elim` takes a substitution S and a list of parameters *(x ...)* as its arguments, and returns a modified substitution *S'* such that the intersection of both the domain and the range of *S'* with *(x ...)* is empty. (Although either may contain terms that contain variables in *(x ...)*). A parameter x is eliminated by `param-elim` by simply dropping the disequation $x \neq t$, if x does not occur as the right or left-hand side of any other equation in S. Otherwise if there are equations *{x $\neq$ t_1, ..., x $\neq$ t_n}*, it is eliminated by replacing those equations with *{$t_i \neq t_j$, ...}*, where $i \leq i,j \leq n$, $i \neq j$. [3]

If after this process S is empty, then DU fails (the first clause), since it is impossible to find a substitution to make s and t different. Otherwise (DU's third clause), a constraint of the form *(∀ (x_a ...) (f x ...) $\neq$ (f t ...))* is added. This is equivalent to the disjunction *(x $\neq$ t)* $\vee$ ..., under the quantifier with parameters x_a ... (which may remain because we have only eliminated them at the top level of the terms t ...).

[3]The `elim-x` metafunction, seen in the specification of `param-elim` in figure 9, implements this find/pair operation.

Finally, if it is ever the case that a constraint in D no longer has at least one disequation $x \neq t$ where the right hand side is a variable, then it is removed and added to the top of the current problem P (DU's final clause). The intuition is that as long as one disequation in a constraint looks like $x \neq t$, where x is not a parameter, it can be satisfied by simply choosing x to be something other than t. Otherwise it may be invalid so it must be checked by applying U and `param-elim` once again.

> **Theorem 2** For any problem P, $\mathsf{DU}(\!|P, (), ()|\!)$ terminates with $\perp$ if the equations in P have no unifier consistent with the disequational constraints in P. Otherwise, it terminates with *(S_Ω : D)* where S_Ω is a most general unifier for the equations in P, and S_Ω is consistent with the disequational constraints in P. The constraints in D are equivalent to the originals in P.

A proof of this theorem is given in Appendix A. This method of solving disequational constraints is based on the approaches detailed in Colmerauer [6] and Comon and Lescanne [8]. Colmerauer [6] shows how to solve disequational constraints by using a unifier to simplify them, as we do here, however his constraints do not include universal quantifiers. Comon and Lescanne [8], on the other hand, show how to solve the more general case of problems of the form $\exists\, x... \;\forall\, y... \;\phi$ where ϕ is a formula consisting of term equalities and their negation, disjunction, and conjunction. They give their solution as a set of rewrite rules, and although their approach will solve the same constraints as ours, the equivalence of the two isn't completely trivial. One advantage of the approach we take is that it can be easily understood and implemented as an extension to the unifier.

3.6. Handling More of the Pattern Language

Up to this point term generation and the constraint solver have been presented using a very simplified version of Redex's internal pattern language. Here the extension of both to handle

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

Figure 10: The subset of the internal pattern language supported by the generator

a more complete subset of the pattern language is discussed. The part of the pattern language actually supported by the generator is shown in figure 10. Racket symbols are indicated by the *s* non-terminal, and the *c* non-terminal represents any Racket constant (which is considered to be equal to itself only by the matcher and the unifier.) The generator is not able to handle parts of the pattern language that deal with evaluation contexts, compatible closure, or "repeat" patterns (ellipses).

The extensions to the pattern language are enumerated by the new[4] productions of the *p* non-terminal in figure 10. We now explain briefly each of the new productions along with the approach used to handle it in the generator.

Named Patterns These corrsepond to variables *x* in the simplified version of the pattern language from figure 5, except now the variable is attached to a sub-pattern. From the matcher's perspective, the `(name s p)` production is intended to match a term with a pattern *p* and then bind the matched term to the name *s*. In the generator named patterns are treated essentially as logic variables. When two patterns are unified, they are both pre-processed to extract the pattern *p* for each named pattern, which is rewritten into a logic variable with the identifier *s*. This is done by finding the value for *s* in the current substitution, and unifying *p* with that value. The result is used

[4]New with respect to figure 5

to update the value of *s* in the current substitution. (If *s* is a new variable, then its value simply becomes *p*).

Built-in Patterns The *b* and *v* non-terminals are built-in patterns that match subsets of Racket values. The productions of *b* are self-explanatory; `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. Equalities between built-in patterns have the obvious relationship; the result of an equality between `real` and `natural`, for example, is `natural`, whereas an equality between `real` and `string` simply fails. As equalities of this type are processed, the values of terms in the current substitution are refined.

The *v* non-terminals match Racket symbols in varying and commonly useful ways; `variable-not-otherwise-mentioned`, 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.

Mismatch Patterns These are patterns of the from `(mismatch-name` *s p*`)` which match the pattern *p* with the constraint that two mismatches 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.

Non-terminal Patterns Patterns of the form `(nt` *n*`)` are intended to successfully match a term if the term matches one of the productions of the non-terminal *n*. (Recall that patterns are always constructed in relation to some language.) It is less clear how non-terminal patterns should be dealt with in the unifier. It would be nice to have an efficient method to decide if the terms defined by some pattern intersected with those defined by some non-terminal, but this reduces to the problem

of computing the intersection of tree automata, which is known to have exponential complexity. [7] Instead a conservative check is used at the time of unification and the non-terminal information is saved.

When a pattern is equated with a non-terminal, the non-terminal is unfolded once by retrieving all of its productions and replacing any recursive positions of the non-terminal with the pattern `any`. The pattern is normalized by replacing all variable positions with `any`. Then it is verified that the normalized pattern unifies with at least one of the abbreviated productions. The check is relatively inexpensive, and the results can be cached. This method is effective at catching cases where a pattern should obviously fail to unify with a non-terminal, but because it may succeed where a more complete method would fail, a later check is necessary.

When a pattern successfully unifies with a non-terminal, the pattern is annotated with the name of the non-terminal in the current substitution. The intention of this is that once a pattern becomes fully instantiated (once it becomes a term), it becomes simple to verify that it does indeed match one of the non-terminal's productions. All annotated non-terminals are verified when result patterns are instantiated as terms.

This approach to handling grammar information in a unifier is somewhat ad-hoc, and it might be interesting to consider more fully how such structure could be used to aid unification.

3.6.1. Instantiating Patterns

The result of the derivation generation process is a pattern that corresponds to the original goal, an environment that corresponds to the substitution generated by the generation process, and a set of disequational constraints. The final step is to perform the necessary random instantiations of pattern variables to produce a term as the result. Variables in the environment will resolve to patterns consisting of `list` constructors, racket constants, non-terminals, and built-in patterns.

The portion of the environment necessary to instantiate the goal pattern is processed to eliminate built-in patterns and non-terminals by using the context-free generation method, and `list` terms are converted to racket lists. Then the disequational constraints are checked for consistency with the new environment. Finally, the goal pattern is converted to a term by using the same process and resolving the necessary variables.

CHAPTER 4

Evaluating the Generator

This chapter describes the evaluation of the new random generator. The generator was evaluated by comparing it to other methods of random generation that can be applied to similar specifications. Generators were compared in terms of their testing effectiveness, measured by how long they took to find known counterexamples.

The first section discusses the comparison of the derivation-based generator with a straightforward approach based on grammars, as originally provided by Redex and explained in Chapter 1. For this comparison, we developed a benchmark of several Redex models, introducing a number of bugs into each, and evaluated the performance of the two generators on each model/bug pair.

We then compared the performance of the derivation-based generator with pre-existing Redex model and test framework for the Racket Virtual Machine, as explained in Section 4.2. This effort had a more refined approach to random generation that post-processed terms generated from the grammar to make them more likely to be valid test cases. In this case we reintroduced several bugs found during the original effort and compared the ability of the two generation approaches to produce counterexamples.

4.1. A Bugfinding Benchmark

To compare the performance of the derivation-based generator to the original grammar-based approach used in Redex, we developed a bugfinding benchmark. The benchmark is not necessarily limited in application to the two approaches we have used it with; it could in principle be used with

any generation method that that takes some part of a semantic model (i.e. a type system, reduction relation, or type system) as its input and generates terms automatically. It couldn't be applied as easily, however, with approaches that are not "push-button" and require some human ingenuity[1] to implement the generator.

The benchmark consists of six different Redex models, each of which consists of a grammar, a dynamic semantics in the form of a reduction relation or metafunction, and some static well-formedness property formulated with judgement forms and metafunctions. (The last is usually a type system.) Finally, each has some predicate that relates the dynamic properties of the system to the static properties. Type soundness, for example, states that reducing a well-typed term that is not a value is always possible and always preserves the type of the term.

For each model, several "mutations" provide the tests for the benchmark. The mutations are made by manually introducing bugs into a new version of the model, such that each mutation is identical to the correct[2] version aside from a single bug. The models used are:

- **stlc** A simply-typed lambda calculus with base types of numbers and lists of numbers, including the constants `cons`, `head`, `tail`, and `nil` (the empty list), all of which operate only on lists of numbers. The property checked is type soundness: the combination of subject reduction (that types are preserved by the reductions) and progress (that well-typed non-values always take a reduction step). 9 different mutations (bugs) of this system are included.
- **poly-stlc** This is a polymorphic version of **stlc**, with a single numeric base type, polymorphic lists, and polymorphic versions of the same constants. Again, the property checked is type soundness. 9 mutations are included.

[1] On the other hand, such approaches have the advantage of being limited only by the ingenuity of the implementor.
[2] So far as we know. The "correct" versions satisfy our unit tests and had no bugs we were able to uncover with random testing.

- **stlc-sub** The same language and type system as **stlc**, except that in this case all of the errors are in the substitution function. Type soundness is checked. 9 mutations are included.
- **list-machine** An implementation of the *list-machine benchmark* described in Appel et al. [1], this is a reduction semantics (as a pointer machine operating over an instruction pointer and a store) and a type system for a seven-instruction first-order assembly language that manipulates `cons` and `nil` values. The property checked is type soundness as specified in Appel et al. [1], namely that well-typed programs always step or halt ("do not get stuck"). 3 mutations are included. This was a pre-existing implementation of this system in Redex that we adapted for the benchmark.
- **rbtrees** A model implementing red-black trees via a judgment that a tree has the red-black property and a metafunction defining the insert operation. The property checked is that insert preserves the red-black tree property. 3 mutations of this model are included.
- **delim-cont** A model of the contract and type system for delimited control described in Takikawa et al. [23]. The language is essentially PCF extended with operators for delimited continuations and continuation marks, and contracts for those operations. The property checked is type soundness. 2 mutations of this model are included, one of which was found and fixed during the original development of the model. This was a pre-existing model developed by Takikawa et al. [23] which was adapted for the benchmark.

For each mutation and each generation method, the benchmark repeatedly generates random terms and tests the correctness property, tracking how long the model runs for before the property is falsified and a counterexample is found. For each mutation/method instance, we ran the tests for increasing intervals of time from 5 minutes up to 24 hours, stopping if the average interval stabilized. The tests were run for a maximum of 48 hours total for each instance. Thus if a

generation method failed to find a counterexample for some mutation, it ran for 2 days without finding a single counterexample.

The results of the benchmark for the grammar-based generator and the derivation based generator are shown in figure 11. In each of these tests, the grammar-based generator is used to generate terms from the grammar which are then discarded if they fail to be well-typed. If they are well-typed, then the correctness property is checked. The derivation-based generator, on the other hand, generates terms that are already well-typed so the correctness property can be checked for every term that it generates.

The relative performance of the two generators is related to two metrics which are not visible in figure 11: the rate at which terms are produced, and the ratio of counterexamples to terms produced. In fact, the grammar-based generator produces terms at rates up to greater than one hundred times as fast as the derivation generator. On the other hand, the ratio of counterexamples to terms produced is *much* better for the derivation generator — if we had used this for our metric, in fact, the the difference between the two would appear to be even greater. The metric used here was chosen to reflect as closely as possible the effective difference between the two approaches in actual application during the development process. However, we note in passing that there could be significant payoff to increasing the speed of the derivation generator, and it seems likely there is much more room for optimization here than in the grammar-based approach.

The averages in figure 11 span nearly 6 orders of magnitude from less than a tenth of a second to almost two hours, and are shown on a logarithmic scale. The error bars shown reflect 95% confidence intervals in the average, meaning there is a 95% chance that the actual average falls within the delineated interval. The derivation-based generator (“search”) succeeded on each of the 34 tests, and the grammar-based generator (“grammar”) succeeded on only 26. As noted before,

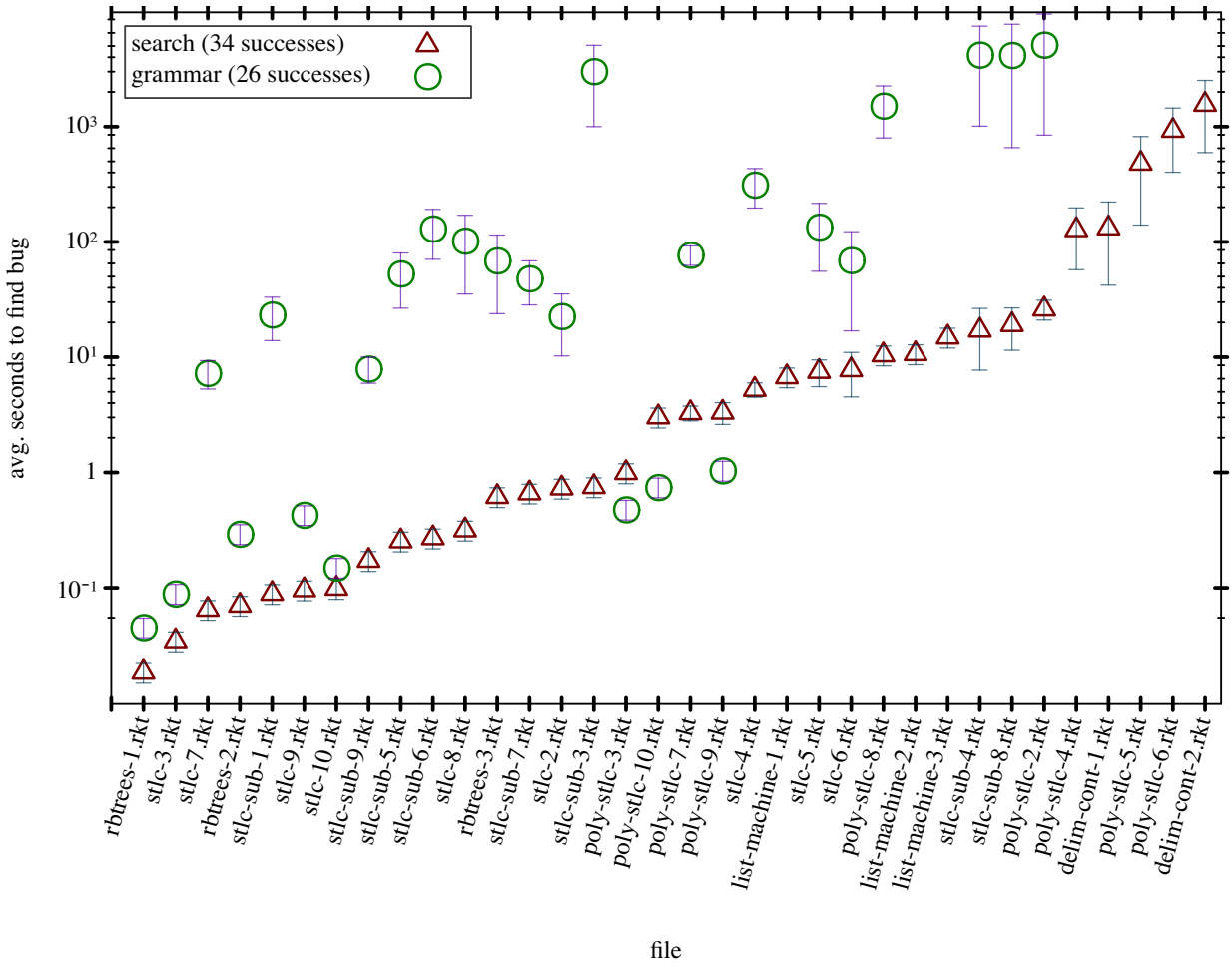


Figure 11: Comparison of the derivation-based generator (‘‘search,’’ the red triangles) and the grammar-based generator (‘‘grammar,’’ the green circles) on the bug-finding benchmark tests. The y-axis is the average time to find the bug in seconds, in a log scale. The error bars are 95% confidence intervals in the average.

for each failure this generator ran for 48 hours without finding a counterexample. The derivation-based generator is consistently faster (with three exceptions), by a factor of up to over 1000 in some cases. The ratio of the two averages varies dramatically, however. In the few cases where the grammar-based generator is faster, both averages are near a second, indicating that these aren’t particularly difficult bugs to find, and the ratio between the two is not large.

4.2. Comparison with an Established Redex Testing Effort

To further evaluate the performance of the derivation-based generator, we compared it with a pre-existing test framework for the Racket virtual machine and bytecode verifier. [20] This testing approach was based on a Redex model of the virtual machine and verifier, and used the approach of generating random bytecode terms and processing them with a "fixing" function that modifies the terms to make them more likely to pass the verifier. This testing approach proved effective, successfully finding more than two dozen bugs in the machine model and a few in the actual Racket implementation.

Figure 12 compares the performance of the derivation generator and the grammar-based generator with a "fixing" function. The derivation generator is used to directly produce bytecode that will pass the verifier. The other generation method is that used by Klein et al. [20] and uses Redex to automatically produce terms satisfying the byte-code grammar, which are then passed to the fixing function. The main changes performed by the fixing function are to adjust stack offsets to agree with the current context and replace some references with values, greatly increasing the chance that the randomly generated bytecode will be verifiable.

The two generation methods were compared on 6 bugs discovered during the development of the virtual machine model by Klein et al. [20] and found to be difficult to uncover with their random-testing efforts. (The labels in figure 12 correspond to those from figure 28 in that paper.) Interestingly, using the same generation methods as they did, we were able to find one bug they failed to find in over 25 million attempts, and failed to find one bug they uncovered with random testing. This is yet more evidence of the sensitivity of random testing to seemingly unrelated changes in the model, and the difficulty of repeating specific random results.

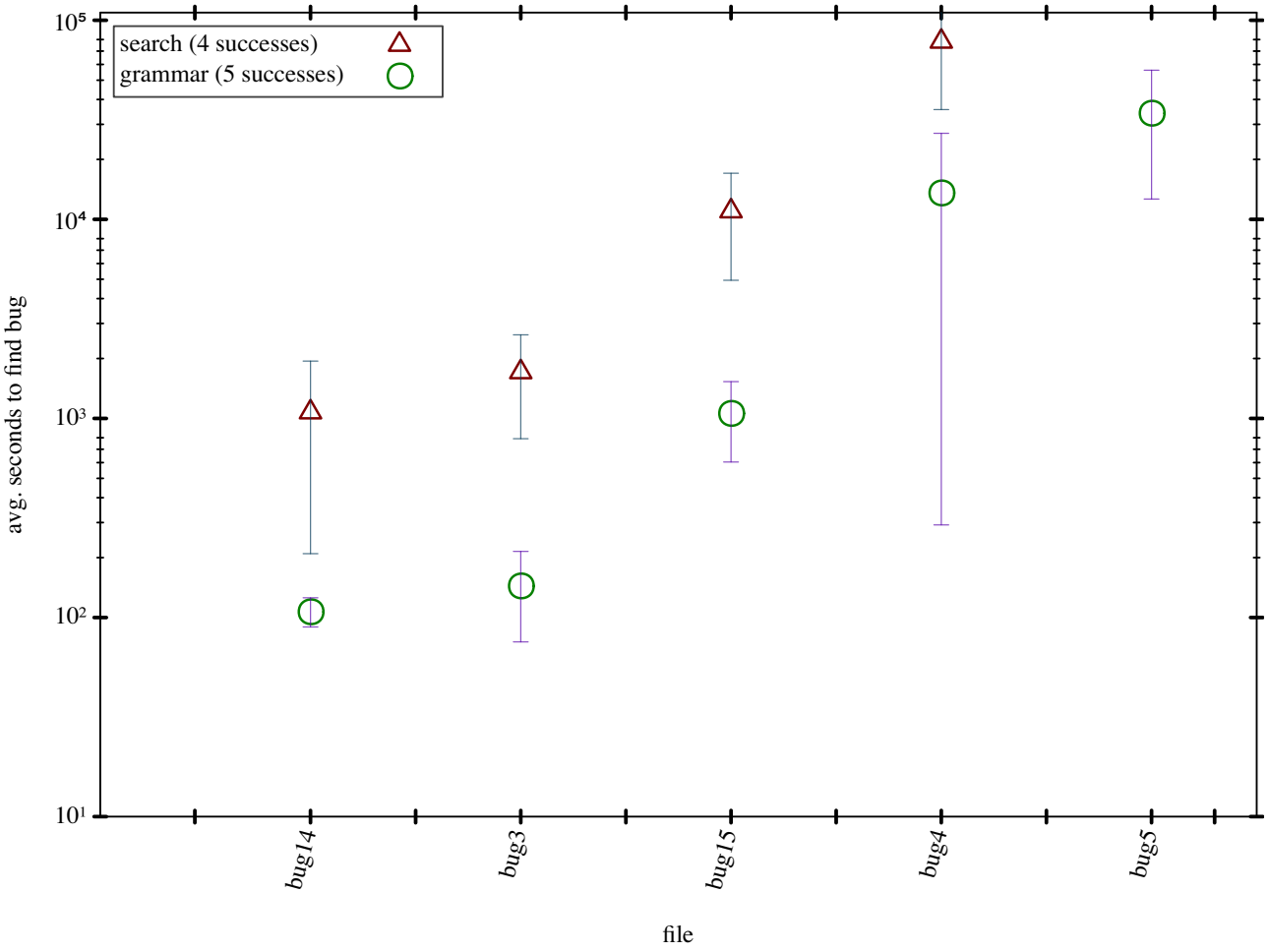


Figure 12: Comparison of bug-finding effectivenesss of the typed term generator vs. the grammar generator and a ‘‘fixing’’ function. The generators were compared on 6 bugs, one of which both failed to uncover. Again, the y-axis is the average time to find a bug in seconds on a log scale, and error bars are 95% confidence intervals in the average.

The results of the comparison are shown in figure 12. These were indeed all difficult bugs to find, with the lowest average time to find a bug being several minutes and the largest close to a full day. (We ran these tests for up to a total of nearly four days for each instance.) The derivation generator is about a factor of ten slower in the four cases where it succeeded. Thus in this case the grammar/fix method is significantly more effective. (There is some reason to believe this specific

case may be especially difficult for the derivation generator and perhaps easier for a fixing function, as discussed below.)

The fixing function used in this case was developed simultaneously with the model and doesn't provide the same "push-button" feedback we hope for from the derivation generator. Writing such functions requires some amount of cleverness on the part of the user and imposes some cognitive overhead on the development process. On the other hand, there are significant constraints (also addressed below) placed on the form of the model to make it compatible with derivation generation.

Finally, we note that the ability of the derivation generator to find these bugs is encouraging in the sense that they indicate it is at least minimally effective when applied to models of production systems, and could be used for testing efforts extending beyond "toy" semantic models.

4.2.1. Translating the model and effects on term generation

Since this effort dealt with a pre-existing Redex model of non-trivial complexity, it provides an opportunity to consider the limitations of the derivation generator from the perspective of the effort needed to translate the model into a form the new generator could handle.

As already noted, the derivation generator isn't able to handle ellipses ("repeat" patterns) or uses of `unquote`. Ellipses provide rich ways of destructuring and constructing lists and `unquote` allows escaping to arbitrary Racket code from inside of Redex. Removing ellipses and `unquote`s from the judgment forms and metafunctions of the verifier followed a process simliar to other efforts to translate pre-existing Redex code into a form the new generator can process. Lists had to be replaced with an explicit representation using pairs, and uses of ellipses to process lists had to be replaced with metafunctions dispatching on cons and nil. In the verifier, uses of `unquote` almost exclusively escaped to Racket to do arithmetic on natural numbers for indexing into the

stack and verifying offsets. This was replaced with an explicit unary representation of arithmetic, and related list operations such as length and indexing.

Besides the pain of forcing users to avoid ellipses and roll their own unary arithmetic operations, these changes can have a significant effect on the performance of the generator. Clearly list operations and unary arithmetic defined using Redex are much slower than Redex's internal list processing or Racket's number operations. Also, the recursive nature of the definitions skews the generator towards producing shorter lists and smaller numbers. In fact, since the generator chooses equally between alternatives, simply using it to produce unary numbers will result in a distribution where the probability of generating a number falls off exponentially with its size.

The specification of the bytecode and verifier prove to be especially difficult for the derivation generator to handle in this case. Interestingly, they make a "fixing" function whose main transformation is to fix stack offsets somewhat easier to write, since that functions main task is to track and adjust a few natural numbers. Fixing functions for properties with richer recursive structure may present more difficulty. In those cases where a good fixing function becomes more like a reimplementation of the type system, using the derivation generator may be a better option.

This comparison also exposes some opportunities to improve the derivation generator. Adding natural number arithmetic and finite domain constraints might make it much more effective for cases like this by avoiding the inefficient and poorly distributed unary representation. Similarly, adding support for repeat patterns would provide gains in both expressiveness and testing effectiveness. (Pattern and sequence unification [21] might be a productive area to investigate in this direction.)

CHAPTER 5

Related Work

Quickcheck [5] is a widely-used library for random testing in Haskell. It provides combinators supporting the definition of testable properties, random generators, and analysis of results. Although Quickcheck's approach is much more general than the one taken here, it has been used to implement a random generator for well-typed terms robust enough to find bugs in GHC. [22] This generator provides a good contrast to the approach of this work, as it was implemented by hand, albeit with the assistance of a powerful test framework. Significant effort was spent on adjusting the distribution of terms and optimization, even adjusting the type system in clever ways. Our approach, on the other hand, is to provide a straightforward way to implement a test generator, ideally by simply writing down the type system, the inherent tradeoff being that an automatically derived generator is almost certain to be much slower and likely to have a less effective[1] distribution of terms. The interesting question is how large this tradeoff is. We hope to provide an answer to this question by investigating a comparison with this work similar to the one we conducted for the Racket Virtual Machine model.

Random program generation for testing purposes is not a new idea and goes back at least to Hanford [13], who details the development and application of the "syntax machine", essentially a tool for producing random expressions from a grammar similar to Redex original term generation method. The tool was intended for testing compilers, a common target for this type of random

[1]Less effective in the sense of less likely to find bugs. The term distribution of Palka [22] is essentially optimized in this way, as much as is possible.

generation. Other uses of random testing for compiler testing throughout the years are discussed in the 1997 survey of Bourjarwah and Saleh [4].

In the area of random testing for compiler, of special note is Csmith [24], a highly effective tool for generating C programs for compiler testing. Csmith generates C programs that avoid undefined or unspecified behavior. These programs are then used for differential testing, where the output of a given program is compared across several compilers and levels of optimization, so that if the results differ, at least one of test targets must contain a bug. Csmith represents a significant development effort at 40,000+ lines of C++ and the programs it generates are finely tuned to be effective at finding bugs based on several years of experience. This approach has been effective, finding over 300 bugs in mainstream compilers as of 2011.

Efficient random generation of program terms has seen some interesting advances in previous years, much of which focuses on enumerations. Feat [9], or "Functional Enumeration of Algebraic Types," is a Haskell library that exhaustively enumerates a datatype's possible values. The enumeration is made very efficient by memoising cardinality metadata, which makes it practical to access values that have very large indexes. The enumeration also weights all terms equally, so a random sample of values can in some sense be said to have a more uniform distribution. Feat was used to test Template Haskell by generating AST values, and compared favorably with Smallcheck in terms of its ability to generate terms above a certain size. (QuickCheck was excluded from this particular case study because it was "very difficult" to write a QuickCheck generator for "mutual recursive datatypes of this size", the size being around 80 constructors. This provides some insight into the effort involved in writing the generator described in Palka [22].)

Another, more specialized, approach to enumerations was taken by Grygiel and Lescanne [12]. Their work addresses specifically the problem of enumerating well-formed lambda terms. (Terms where all variables are bound.) They present a variety of combinatorial results on lambda terms,

notably some about the extreme scarcity of simply-typable terms among closed terms. As a by-product they get an efficient generator for closed lambda terms. To generate typed terms their approach is simply to filter the closed terms with a typechecker. This approach is somewhat inefficient (as one would expect due to the rarity of typed terms) but it does provide a uniform distribution.

Instead of enumerating terms, Kennedy and Vytiniotis [???] develop a bit-coding scheme where every string of bits either corresponds to a term or is the prefix of some term that does. Their approach is quite general and can be used to encode many different types. They are able to encode a lambda calculi with polymorphically-typed constants and discuss its possible extension to even more challenging languages such as System-F. This method has the potential to be used for random generation by decoding randomly chosen bit strings.

CHAPTER 6

Conclusion

We extended the random testing capabilities of PLT Redex to support the generation of terms satisfying judgment forms and metafunctions. The previous method was an approach based on recursively unfolding the non-terminals of a grammar. The new method outperformed the old on a benchmark of bugs we developed, but a hand-written generator based on the old method outperformed the new at finding bugs in a model of the Racket virtual machine.

Random generation based on recursively expanding productions in a grammar with no additional processing or heuristics is really the simplest possible "pushbutton" strategy, so performing better than this approach should be considered only an initial hurdle for any alternative strategy. As such, it is encouraging that our derivation-based generation strategy performed much better than the grammar generator on the benchmark, but that doesn't mean we should necessarily embrace this as a better strategy. Indeed, our experience with the virtual machine model shows that post-processing terms generated from a grammar can perform much better than derivation based generation for at least one system. (Although, as we point out in Chapter 4, there are some specific characteristics of this system that are difficulties for the derivation generator.) The approach of using the grammar generator with a fixing function is more fully developed, and we expect to be able to improve the derivation generator as we gain more experience with it.

The benchmark we developed for comparing the grammar-based generation strategy to the derivation-based strategy is a valuable foundation for the evaluation of future work on random

testing. There are a variety of possible strategies that, if implemented successfully, could be evaluated immediately with this benchmark. For example, it may be possible to generate well-typed terms from judgment form definitions using approaches similar to those of Duregard et al. [9] or Kennedy and Vytiniotis [???]. If we are able to implement such an idea, the benchmark we already have could be used to compare it to both of Redex's existing generation methods.

The benchmark itself should still be considered a work in progress. Extending it with a further variety of systems could prove illuminating. Possibilities include adding type systems for imperative programming languages, or systems based on pre-existing models with realistic bugs. We plan to make the benchmark more modular and work to extend it with new bugs and systems when appropriate. As we continue to develop the benchmark we hope that it will prove more and more useful in improving Redex's random testing capabilities.

In terms of generating well-typed terms, there are many alternative strategies and variations and improvements on the strategy that we have used that remain to be explored. We believe that the results obtained with our method make a good argument for continuing with those investigations.

This work is certainly not the final word in random generation based on rich properties such as type judgments. However, it does demonstrate the potential of such an approach, both in terms of its utility and feasibility. We hope that it can serve as a basis for more valuable work to be done in this area.

Bibliography

[1] Andrew W. Appel, Robert Dockins, and Xavier Leroy. A list-machine benchmark for mechanized metatheory. *Journal of Automated Reasoning* 49(3), pp. 453–491, 2012.

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

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

[4] Abdulazeez S. Bourjarwah and Kassem Saleh. Compiler test case generation methods: a survey and assessment. *Information & Software Technology* 39(9), pp. 617–625, 1997.

[5] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 268–279, 2000.

[6] Alain Colmerauer. Equations and Inequations on Finite and Infinite Trees. In *Proc. Intl. Conf. Fifth Generation Computing Systems*, pp. 85–99, 1984.

[7] 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`

[8] Hubert Comon and Pierre Lescanne. Equational Problems and Disunification. *Journal of Symbolic Computation* 7, pp. 371–425, 1989.

[9] Jonas Duregard, Patrik Jansson, and Meng Wang. Feat: Functional Enumeration of Algebraic Types. In *Proc. ACM SIGPLAN Haskell Wksp.*, pp. 61–72, 2012.

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

[11] 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.

[12] Katarina Grygiel and Pierre Lescanne. Counting and generating lambda terms. *J. Functional Programming* 23(5), pp. 594–628, 2013.

[13] Kenneth V. Hanford. Automatic Generation of Test Cases. *IBM Systems Journal* 9(4), pp. 244–257, 1970.

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

[15] Krystof Hoder and Andrei Voronkov. Comparing Unification Algorithms in First-Order Theorem Proving. In *Proc. KI 2009: Advances in Artificial Intelligence*, pp. 435–443, 2009.

[16] 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.

[17] Andrew J. Kennedy and Dimitrios Vytiniotis. Every bit counts: The binary representation of typed data and programs. *J. Functional Programming* 22, pp. 529–573.

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

[19] 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.

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

[21] Temur Kutsia. Unification with Sequence Symbols and Flexible Arity Symbols and Its Extension with Pattern-Terms. In *Proc. Intl. Conf. Artificial Intelligence, Automated Reasoning, and Symbolic Computation*, pp. 290–304, 2002.

[22] Michal H. Palka. Testing an Optimising Compiler by Generating Random Lambda Terms. Licentiate dissertation, Chalmers University of Technology, Göteborg, 2012.

[23] Asumu Takikawa, T. Stephen Strickland, and Sam Tobin-Hochstadt. Constraining Delimited Control with Contract. In *Proc. Euro. Symp. Programming*, pp. 229–248, 2013.

[24] Xuejin Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *Proc. ACM Conf. Programming Language Design and Implementation*, pp. 283–294, 2011.

APPENDIX A

Proof of Theorem 2

For a substitution $\sigma = \{x \mapsto t \dots\}$, $Range(\sigma) = \{x \dots\}$, $Domain(\sigma) = \{t \dots\}$. We write $\forall \vec{x} \phi$ to indicate $\forall x_1 \forall x_2 \dots \forall x_n \phi$ where $\vec{x} = \{x_1, x_2, \dots, x_3\}$. We write $[\sigma]$ for $\{x = t, \dots\}$, the equational representation of σ.

A substitution σ is idempotent if $\sigma\sigma = \sigma$, equivalently if $Range(\sigma) \cap Variables(Domain(\sigma)) = \emptyset$. We write $U[s = t, ...]$ to mean unifying the equations $s = t, ...$, the equivalent of applying the function U from Chapter 2 with the problem $s = t, ...$ and an empty substitution and disequational constraint set. We make use of the fact that U always returns an idempotent substitution if it succeeds.

Lemma 1. *If σ is idempotent then $\theta = \theta\sigma \Leftrightarrow \sigma \leq \theta$.*

PROOF. The forward direction holds by definition. For the reverse direction, by definition there must be some ψ such that $\psi\sigma = \theta$, so for any x,

$$\theta\sigma x = \psi\sigma\sigma x = \psi\sigma x = \theta x$$

where the middle equality depends on the idempotency of σ. Since they agree on all variables, $\theta = \theta\sigma$. □

Lemma 2. *If $U[s, t] = \gamma = \{x_1 \mapsto t_1 \dots x_n \mapsto t_n\}$, then for any unifier θ of s and t, $\theta x_i = \theta t_i$, for any $1 \leq i \leq n$.*

PROOF. Since $\gamma \leq \theta$ and γ is idempotent, $\theta = \theta\gamma$ (Lemma 1), so $\theta x_i = \theta\gamma x_i = \theta t_i$. □

Lemma 3. *For* $U[s = t] = \{x_1 \mapsto t_1 \ldots x_n \mapsto t_n\}$,

$$U[\omega s = \omega t] = \bot \Leftrightarrow U[\omega(x_1 \ldots x_n) = \omega(t_1 \ldots t_n)] = \bot$$

PROOF. For the forward direction, by definition there is no ϕ such that $\phi\omega s = \phi\omega t$. Now suppose there is some ρ such that $U[\omega(x_1 \ldots x_n) = \omega(t_1 \ldots t_n)] = \rho$. Using the fact that $\omega(x_1 \ldots x_n) = (\omega x_1 \ldots \omega x_n)$ and $\omega(t_1 \ldots t_n) = (\omega t_1 \ldots \omega t_n)$ along with $U[(\omega x_1 \ldots \omega x_n) = (\omega t_1 \ldots \omega t_n)] = U[\omega x_1 = \omega t_1, \ldots, \omega x_n = \omega t_n]$, it's clear that for any i, $\rho\omega x_i = \rho\omega t_i$. This gives us $\rho\omega\gamma x_i = \rho\omega t_i = \rho\omega x_i$, and implies that $\gamma \leq \rho\omega$, so $\rho\omega s = \rho\omega t$, a contradiction. Thus there can be no such ρ and $U[\omega(x_1 \ldots x_n) = \omega(t_1 \ldots t_n)] = \bot$.

For the reverse direction, distributing the substitution and the decomposition as before results in $U[\omega x_1 = \omega t_1, \ldots, \omega x_n = \omega t_n] = \bot$, which means that for some i, $U[\omega x_i = \omega t_i] = \bot$ and there is no ϕ such that $\phi\omega x_i = \phi\omega t_i$. Since any unifier ρ of s and t must satisfy $\rho x_i = \rho t_i$ for all i (Lemma 2), this implies that there is no unifier of ωs and ωt, so $U[\omega s = \omega t] = \bot$. □

We are interested in the validity of the constraint $\forall\vec{x}s \neq t$, with respect to some substitution σ, or the validity of $\forall\vec{x}\sigma s \neq \sigma t$. ($\sigma$ corresponds to the current substitution in the constraint solver, so this corresponds to maintaining consistency between the substitution created by the equational constraints and the disequational constraint.) We consider $\forall\vec{x}\sigma s \neq \sigma t$ to be consistent with σ so long as there is some substitution ω such that $\forall\vec{x}\omega\sigma s \neq \omega\sigma t$ is true. Note that both σ and ω are only substitutable in the constraint if they don't contain any variables bound by the quantifier, so we must require (without loss of generality) that $Variables(\sigma) \cap \vec{x} = \emptyset$ and $Variables(\omega) \cap \vec{x} = \emptyset$. Now note that

$$\forall\vec{x}\omega\sigma s \neq \omega\sigma t \Leftrightarrow \neg\exists\vec{x}\omega\sigma s = \omega\sigma t$$

If $\exists\vec{x}\omega\sigma s = \omega\sigma t$, then we can construct a substitution $\sigma_{\vec{x}}$ such that $\sigma_{\vec{x}}\omega\sigma s = \sigma_{\vec{x}}\omega\sigma$, i.e. there is a unifier ($\sigma_{\vec{x}}$) for $\omega\sigma s$ and $\omega\sigma t$. The negation means there cannot be such a unifier, so

$$\neg\exists\vec{x}\omega\sigma s = \omega\sigma t \Leftrightarrow U[\omega\sigma s, \omega\sigma t] = \bot$$

and as long as such an ω can be constructed, the constraint $\forall\vec{x}s \neq t$ is still consistent with the substitution σ. We now show how the results of the unification $U[s,t]$ can be used to check for the existence of ω.

In what follows, $U[s = t] = \gamma = \{x_1 \mapsto t_1 \ldots x_n \mapsto t_n\}$, and we are considering the existence of some ω given some σ so that $U[\omega\sigma s = \omega\sigma t] = \bot$. Both ω and σ are subject to the restriction that $Variables(\omega) \cap \vec{x} = \emptyset$ and $Variables(\sigma) \cap \vec{x} = \emptyset$ for some set of variables $\vec{x}$. ($\vec{x}$ corresponds to the universally quantified variables of the constraint.) From Lemma 3, we know that if $U[\omega\sigma s = \omega\sigma t] = \bot$, then $U[\omega\sigma x_1 = t_1, \ldots, \omega\sigma x_n = \omega\sigma t_n] = \bot$, or that for some i, $U[\omega\sigma x_i = \omega\sigma t_i] = \bot$; we call $x_i = t_i$ (or $\{x_i \mapsto t_i\}$) the part of γ that is excluded by ω. First we prove some lemmas that allow us to restrict the excluded part of γ.

Lemma 4. *If $\gamma = \{x \mapsto t_x\}\gamma'$ and $x \in \vec{x}$, then ω excludes $[\gamma]$ iff ω excludes $[\gamma']$.*

PROOF. Because γ is idempotent, x does not occur elsewhere in γ. The variable restriction means that $\omega\sigma x = x$, and since x is otherwise unrestricted, it can never be the case that $U[x = \omega\sigma t_x] = \bot$. Thus if ω excludes $[\gamma]$, it must exclude $[\gamma']$. Clearly if ω already excludes $[\gamma']$, it excludes $[\{x \mapsto t_x\}\gamma']$. □

Lemma 5. *If $\gamma = \{y_1 \mapsto x\} \ldots \{y_n \mapsto x\}\gamma'$, where $y_i \notin \vec{x}$ and $x \in \vec{x}$, then ω excludes $[\gamma]$ iff ω excludes $\{y_i = y_j | 1 \leq i, j \leq n\} \cup [\gamma']$.*

PROOF. ω may exclude $[\{y_1 \mapsto x\} \ldots \{y_n \mapsto x\}] = \{y_1 = x, \ldots, y_n = x\}$, if $U[\omega\sigma y_i = \omega\sigma y_j] = \bot$, for any $1 \leq i, j \leq n$, since there is then no valid value for $\omega\sigma x$. This is equivalent to excluding an element of the set of equations $\{y_i = y_j | 1 \leq i, j \leq n\}$. Thus excluding $[\gamma]$ is equivalent to excluding $\{y_i = y_j | 1 \leq i, j \leq n\} \cup [\gamma']$. □

Lemma 6. *If $\gamma = \{y \mapsto x\}\gamma'$, where $y \notin \vec{x}$ and $x \in \vec{x}$ and x does not occur anywhere in γ', then ω excludes $[\gamma]$ iff ω excludes $[\gamma']$.*

PROOF. Analagous to that of Lemma 4. □

Lemma 7. *If $[\gamma]$ has at least one element $x_i = t_i$ such that $\sigma x_i = x_\sigma$, $\sigma t_i \notin \vec{x}$, and $x_\sigma \notin \vec{x}$, then there is an ω that excludes $[\gamma]$.*

PROOF. Simply choose ω such that $U[\omega x_\sigma = \omega\sigma t_i] = \bot$. It will always be possible to do so because x_σ is otherwise unrestricted. □

The strategy used by the function DU is to first apply the transformation of Lemma 3 to get some set of equations ψ, and then apply the trasformations of Lemmas 4, 5 and 6 until ψ has a form that can be checked using Lemma 7.

Lemma 8. *Given some set of equations ψ and ω restricted by the varibles $\vec{x}$, ω excludes ψ iff ω excludes the result of param-elim$[\![\psi, \vec{x}]\!]$.*

PROOF. The first case of param-elim applies the transformation of Lemma 4, and the second case applies the transformations of Lemmas 5 and 6. The other cases don't change ψ. In all cases the property that ω excludes the new set of equations is preserved. □

Lemma 9. *DU preserves the consistency of disequations in P and D with the current substitution or fails, if a disequation is no longer consistent with the current substitution.*

PROOF. The first three cases process the first constraint in P, of the form $\forall \vec{x} s \neq t$. (At this point the current substitution has already been applied to the constraint). First $U[s = t] = \gamma$ is computed to get the set of equations $\psi = [\gamma]$. Then param-elim is applied to ψ to get ψ'. That these operations preserve consistency follows from Lemmas 3 and 8. Now we examine the cases of DU in order.

Case 1. ψ' is empty. The unification $U[s = t]$ cannot be excluded, so the constraint cannot be satisfied and DU fails with $\perp$.

Case 2. $U[s = t]$ failed in the first place. There is no unifier for s and t, so the constraint will always be satisfied, thus it is dropped.

Case 3. ψ' has a form such that currently every equation in it agrees with Lemma 7, so it is currently satisfiable. Lemma 3 is used to combine the equations in ψ' into a single disequational constraint, which is added to D. Lemmas 3 and 8 guarantee the current substitution is consistent with the new constraint iff it is consistent with the original.

Case 4. If any constraint in D no longer satisfies the check of Lemma 7, it is added again to P, where its consistency will again be checked as above. When the current substituion is changed, it is applied to the constraints in D (the last case of the function U), so the action of Case 4 ensures that all constraints remain consistent. □

Lemma 10. *DU always terminates.*

PROOF. Define the lexicographic ordering $\langle n_1, n_2, n_3, n_4 \rangle$ on the P and D arguments to DU, where:

n_1 = the number of variables in P and D that are not universally quantified

n_2 = the number of unsimplified constraints in D, where a simplified constraint has a non-quantified variable in at least one subterm of the left-hand side. (A simplified constraint meet the consistency criteria of Lemma 7.)

n_3 = the number of symbols in P

n_4 = the number of equations in P of the from $t = x$ where t is not a variable

This ordering is well-founded, and decreases with every recursive call in DU. □

Theorem 2 follows immediately from Theorem 1, Lemma 9 and Lemma 10.