NORTHWESTERN UNIVERSITY

Automated Testing for Operational Semantics

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Electrical Engineering and Computer Science

By

Burke Fetscher

EVANSTON, ILLINOIS

December 2015

ABSTRACT

Automated Testing for Operational Semantics

Burke Fetscher

In this dissertation, I investigate the effectiveness of automatic property-based testing in a lightweight framework for semantics engineering. The lightweight approach provides the benefits of execution, exploration, and testing early in the development process, so bugs can be caught early, before significant effort is expended on proofs of correctness. Specifically, I show how lightweight specifications can be leveraged to automatically derive effective test-case generators.

This work is done in the context of PLT Redex, a lightweight semantics framework embedded in Racket. Redex emphasizes property-based testing by allowing users to write predicates expressing desirable properties and attempting to falsify them by automatically generating test cases. In keeping with the lightweight approach, Redex generators are intended to be as "push-button" as possible, and are derived from Redex models with little additional input from the user. I present several methods for deriving generators, including a generic method for randomly generating well-typed expressions, the main contribution of this work. Starting from a specification of a typing judgment in Redex, this method uses a specialized solver that employs randomness to find many different valid derivations of the judgment form.

To evaluate the effectiveness of the different generators, I present a random testing benchmark of Redex models and bugs. I discuss the benchmark and the performance of the different generators at bug-finding, along with an additional case study comparing the typed generator against the best available, custom well-typed term generator. The new generator is much more effective than generation techniques that do not explicitly take types into account and is worse than, but competitive with the custom generator, even though the custom generator is specialized to a particular type system and Redex's generator is generic.

Acknowledgments

It was quite a stroke of luck for me to walk into Robby Findler's office one day five years ago. Undoubtedly, without his encouragement I would not have begun this process, and without his advice and guidance I certainly would not have finished.

Throughout my time at Northwestern, it has been a pleasure to work alongside many many knowledgeable and enjoyable people. Casey Klein was very helpful in introducing me to Redex, and providing the initial prototype that led to this dissertation. James Swaine, Spencer Florence, Vincent St-Amour, Dan Feltey, and Jesse Tov have all been great colleagues from whom I have learned a lot.

Finally, thanks to my family for their support and patience. My parents, for believing in me in spite of it all throughout the years. Most importantly, to my wife Maureen and son Nevin, for being the ones I believe in.

Thanks to the National Science Foundation for supporting this research.

Table of Contents

CHAPTER 1

Introduction

Computer scientists have many tools for understanding programming languages, developed over years of research. Typically those tools were originally developed along with and applied to small language models, calculi that could fit on a few pages of paper or a whiteboard. Since the models themselves are written in a formal language, mechanized tools supporting semantics development are a natural next step, and have been a long-standing research goal. This dissertation investigates the combination of lightweight support for such mechanization with property-based testing, an approach to testing that proves to be particularly effective for semantics engineering.

Lightweight mechanization is distinguished by providing support for executable definitions, and perhaps associated tools, but requiring little effort beyond defining the model. More powerful tools, in contrast, enable machine-checked proofs of soundness properties, but developing such proofs requires more work; writing down definitions is only the beginning of the process.

Lightweight mechanization can be considered the "scripting langauge" approach to engineering a semantics, favoring rapid prototyping and testing as opposed to more powerful analysis or verification. It provides the benefits of executability and testing with low investment. Low investment means changes are low cost, so development can be incremental and iterative. PLT Redex, the framework for which the research in this dissertation was conducted, attempts to provide as many benefits of mechanization as possible while minimizing development effort.

In the end, testing and other forms of automated but non-exhaustive checking may not be enough to provide full confidence that a model is correct, at which point definitions can be ported into a more powerful tool for verification. Such tools typically require more investment to produce an executable model, and the value of using a lightweight tool as a complement is to provide access to the benefits of executability early in the development process.

Unit testing is already a valuable application of lightweight mechanization, but an even more effective approach to semantics development is *property-based* testing. In property-based testing, instead of defining inputs and expected results to a program, a tester formulates a property that should hold over a certain domain. Elements from the domain are then generated automatically, attempting to falsify the property. Since in the long run developers of semantics usually wish to prove specific properties of a system, good testable properties for semantics models are easy to formulate. The other necessary ingredient for effective property-based testing is a good test-case generator, and this dissertation provides evidence that such generators can be automatically derived from lightweight semantics models.

The thesis of this dissertation is:

Lightweight mechanization and automated property-based testing
are effective for semantics engineering.

To support this thesis, I show how lightweight definitions for a semantics can be leveraged to automatically derive test-case generators that effectively expose counterexamples when applied to representative Redex models. I discuss three ways to derive such generators. To show that that property-based testing using the generators is effective, I explain the development of an automated testing benchmark for semantics, consisting of representative Redex models and realistic bugs. I then report on the results of a careful comparison of all Redex's generation methods using the benchmark, as well as a comparison of its most successful method against the best-known, customized generator for well-typed term.

To begin, Chapter 2 introduces operational semantics and Redex in brief by working through the development of a semantics for a small functional language, followed by a discussion of how to mechanize and test the model in Redex. Following that, I discuss the approaches to test-case generation used by Redex. Chapter 3 introduces two approaches to generation based on regular-tree grammars: ad-hoc recursive generators and enumerations. An alternative approach that searches for random derivations satisfying relation and function definitions is introduced in Chapter 4 with an example, and is formally specified and discussed in depth in Chapter 5. Chapter 6 discusses the development of a benchmark intended for comparative evaluation of automated testing methods. The different test generation methods used by Redex are compared using the benchmark in the first section of Chapter 7, and the second section compares the derivation generator to a similar but more specialized generator. Finally, Chapter 8 discusses related work and Chapter 9 concludes.

CHAPTER 2

Operational Semantics and PLT Redex

This chapter provides background in operational semantics and how it is modeled in Redex. It is by no means a comprehensive or systematic summary of either topic, but is intended to explain just enough to understand the rest of the dissertation and show how lightweight semantics engineering works. It begins with an introduction to reduction semantics, the type of operational semantics used by Redex, in section 2.1, by working through the step-by-step development of a semantics for a simple functional language. Then section 2.2 shows how the same language can be coded and run as Redex model that is comparable in size and concision to the pencil and paper semantics. Finally, section 2.3 demonstrates the application of Redex's property-based testing tools to the model.

2.1. Operational Semantics by Example

This section works through the development of a semantics for a simple functional language to illustrate the process of semantics engineering along with reduction semantics, the approach to modeling that Redex is designed for.

Figure 1 shows the grammar for the language we'll be modeling in this section. It is a parenthesized, prefix-notation language of numbers and functions, with two binary operations on numbers, addition (+) and subtraction (-), along with a conditional (if0) that dispatches on whether or not its first argument evaluates to 0 or not. Expressions beginning with λ construct functions of a single argument, which are applied via parenthesized juxtaposition as in Racket or other languages in the

```
e ::= (e e)
    | (λ [x τ] e)
    | (rec [x τ] e)
    | (if0 e e e)
    | (o e e)
    | x | n
n ::= number
o ::= + | -
x ::= variable-not-otherwise-mentioned
```

Figure 1: Grammar for expressions.

Lisp family. Finally, rec expressions support the construction of recursive bindings. Since this is a typed langauge, both of the bindings form also refer to types τ, which are defined later in this section.

A semantics for a programming language is a function from programs to answers. The way the function is defined varies, depending on the intended use of the semantics. Here we will develop an operational semantics in the form of a syntactic machine that transforms programs until they become answers, meaning the domain and range of the function are abstract syntax trees defined by the grammar in figure 1, and it is defined in terms of relations on syntax.

To develop a semantics for this language, we start by identifying the answers, a subset of expressions that are *values*, the results or final states of *computations*. For this language the right choices are numbers and functions, both of which cannot be further evaluated without being used in another expression. We denote values with the addition of another nonterminal, v:

$$v ::= (\lambda\ [x\ \tau]\ e) \mid n$$

We expect that all valid programs (more will be said below about validity) either are a value, or will eventually evaluate to a value.

To this end, we develop a set of relations, pairing any expression in the language that is not a value with another expression that is in some sense “closer” to being a value. (“Closer” in this sense is usually fairly intuitive to a programmer, but in the end it is necessary to prove that a semantics based on these rules does the right thing by eventually transforming valid and terminating programs into values.) For example, the notion of reduction for our binary operations looks like:

$$(\mathrm{o}\ \ulcorner n_1 \urcorner\ \ulcorner n_2 \urcorner) \longrightarrow \ulcorner n_1\ o\ n_2 \urcorner\ \ [\delta]$$

meaning that when a binary operation is applied to two numbers in an expression, we can relate that expression to the number that is the result of the corresponding operation on numbers. (The Gödel brackets $\ulcorner \cdot \urcorner$ lift natural numbers into the syntax of the language.) This allows us to “reduce” such a binary operation to a value. The expression on the left is called the reducible expression or redex, and the expression on the right is called the contractum. A simple example is:

$$(+\ 1\ 2) \longrightarrow 3$$

The rule for function application is more interesting. It says that when a function is applied to a value, the resulting expression is constructed by substituting the value v for all instances of the variable x bound by the function in the function’s body e:

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

where the notation $e\{x \leftarrow v\}$ means to perform capture-avoiding substitution[1] of e for x in v. For example, the application of a function that adds one to its argument to two takes a step as follows:

$$((\lambda\ (x\ \mathsf{num})\ (+\ 1\ x))\ 2) \longrightarrow (+\ 1\ 2)$$

[1]Capture-avoiding substitution (Felleisen et al. 2010) avoids unintentional variable bindings (captures) that can occur when substituting underneath binders by renaming variables appropriately.

$$
\begin{array}{rll}
((\lambda\ [x\ \tau]\ e)\ v) \longrightarrow e\{x \leftarrow v\} & & [\beta] \\
(\mathsf{rec}\ [x\ \tau]\ e) \longrightarrow e\{x \leftarrow (\mathsf{rec}\ [x\ \tau]\ e)\} & & [\mu] \\
(\mathsf{if0}\ 0\ e_1\ e_2) \longrightarrow e_1 & & [\text{if-0}] \\
(\mathsf{if0}\ n\ e_1\ e_2) \longrightarrow e_2 & & [\text{if-n}] \\
\text{where } n \neq 0 & & \\
(\mathrm{o}\ \ulcorner n_1 \urcorner\ \ulcorner n_2 \urcorner) \longrightarrow \ulcorner n_1\ o\ n_2 \urcorner & & [\delta]
\end{array}
$$

Figure 2: Single step reduction, the union of all the notions of reduction for this language.

The complete set of reductions adds rules for if0 and rec and is shown in figure 2. The if0 rule reduces to the second or third argument, depending on the value of the first, and the rec rule unfolds a recursive binding once, substituting the entire expression in the body.

The set of reductions shown in figure 2 capture the notions of computation we intend for our language, but they aren't enough to build an evaluator for all programs, because they only apply at the top level of a term. For example, the term (+ 1 (+ 2 3)) can't be reduced using the δ rule, because at the top level, the second expression is not a number.

To create a relation upon which we can base an evaluator, we need to extend the set of reductions to apply deeper inside of terms. One way to do this is to take the *compatible closure* of the reductions over expressions, which constructs a relation that allows the reductions to be applied anywhere inside a term. This is useful as the basis for an equational calculus, but it is not an evaluator because a given term can be reduced many different ways and evaluators have a fixed strategy.

Instead we can construct a relation that relates each term that can take a step to exactly one term. To do this we use an *evaluation context*, an expression that includes a "hole", denoted by []. This allows a term to be decomposed into a context and, in the hole of the context, a redex. The contractum of the redex can be plugged back into the hole, expressing a single step of computation. The evalutation contexts for our language are denoted by the *E* non-terminal:

$$\begin{aligned} E ::= {} & (E\ e) \mid (v\ E) \mid (o\ E\ e) \mid (o\ v\ E) \\ & \mid (\mathsf{if0}\ E\ e\ e) \mid [] \end{aligned}$$

The first two productions allow reductions to apply on the left-hand side of an application, and on the right right-hand-side of an application if the left-hand-side is a value. The contexts for binary operations are analagous to those for applications, the second to last production allows computation in the condition position of if0 expression, and the last is the hole, which may contain any term.

To construct a standard reduction relation, which we denote with $\longmapsto$, we take the *contextual closure* of the the one-step reduction over *E*:

$$\frac{e_1 \longrightarrow e_2}{E[e_1] \longmapsto E[e_2]}$$

meaning that if a term can take a step according to the one-step reduction, then a context with that same term in its hole can take a step to a term where the corresponding contractum is plugged back into the context at the same position the redex occupied. The intention of the standard reduction is to allow each program to take a step of computation in exactly one way. It may not be immediately obvious from the structure of evaluation contexts that we have this property, so we might wish to test it and, later, prove it. (I address how to test it in Redex in the next section.)

The idea of evaluating a program *e* corresponds to the reflexive transitive closure of the standard reduction, denoted by $\longmapsto^*$. We can define an evaluator in terms of this relation, as follows:

$$\begin{aligned} &\mathit{Eval} : e \rightarrow n \text{ or } \mathsf{function} \\ &\mathit{Eval}[\![e]\!] = n \quad \text{where } e \longmapsto^* n \\ &\mathit{Eval}[\![e]\!] = \mathsf{function} \quad \text{where } e \longmapsto^* (\lambda\ (x\ \tau)\ e_3) \end{aligned}$$

The idea behind Eval is to reduce a program over and over according to the standard reduction until it becomes a value. If the value is a number, we consider that to be an answer. If it is syntax for an unapplied function, we return function, since that syntax really represents an internal state of the evaluator and is not useful.

Note, however, that Eval is not a total function, for several reasons. First, not all programs terminate. (Equivalently, the transitive-reflexive closure of the standard reduction doesn't relate them to values.) Second, some programs may get "stuck", or terminate in expressions that are not values and cannot take another step.[2] We can't avoid the first issue without seriously handicapping our language, but we can tackle the second with a type system, which allows us to separate programs that will get stuck from those that will not.

The type system accomplishes this by categorizing expressions according to what sort of values they will evaluate to. To start, we need a language of types, denoted by τ:

$$\begin{aligned}\tau ::= &\ (\tau \rightarrow \tau)\\ |&\ \mathsf{num}\end{aligned}$$

expressing that we expect two types of values, numbers (num), and functions from one type of value to another, represented by arrows. We can already see that the type system excludes some programs that may not get stuck, namely functions that may return more that one type depending on their input. We could capture functions like this by extending our language of types, but in general the type system must be conservative, excluding some "good" programs in order to exclude all "bad" ones.

We construct the type system using a set inference rules called a typing judgment that defines a relation between a type environments (Γ, to be defined shortly), expressions, and types. As an example, the rule for binary operations is:

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

[2]Another issue sidestepped here that comes up in all real programming languages is that some primitives, such as division, are partial functions.

$$\frac{}{\Gamma \vdash n : \mathsf{num}} \qquad \frac{\tau = \mathsf{lookup}[\![\Gamma, x]\!]}{\Gamma \vdash x : \tau} \qquad \frac{(x\ \tau\ \Gamma) \vdash e : \tau}{\Gamma \vdash (\mathsf{rec}\ [x\ \tau]\ e) : \tau}$$

$$\frac{\Gamma \vdash e_1 : (\tau_2 \to \tau) \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1\ e_2) : \tau} \qquad \frac{(x\ \tau_x\ \Gamma) \vdash e : \tau_e}{\Gamma \vdash (\lambda\ [x\ \tau_x]\ e) : (\tau_x \to \tau_e)} \qquad \frac{\begin{array}{c}\Gamma \vdash e_0 : \mathsf{num} \\ \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau\end{array}}{\Gamma \vdash (\mathsf{if0}\ e_0\ e_1\ e_2) : \tau}$$

Figure 3: The definition of the typing judgment.

expressing that two expressions that evaluate to numbers can be combined using a binary operation, and the resulting expression will evaluate to a number. The relation is defined recursively. To deal with substitutions that occur during evaluation, the type judgment uses the type environment Γ, an accumulator to keep track of the types assigned to variables:

$$\Gamma ::= (x\ \tau\ \Gamma) \mid \bullet$$

The rule for function definition says that if the body of the function has some type with respect to the environment extended with the type of the parameter, then the function itself has an arrow type from the type of the parameter to the type of the body, in the original environment:

$$\frac{(x\ \tau_x\ \Gamma) \vdash e : \tau_e}{\Gamma \vdash (\lambda\ [x\ \tau_x]\ e) : (\tau_x \to \tau_e)}$$

The corresponding rule for typing a variable just looks for the type in the environment. The complete definition of the typing judgment is shown in figure 3. These particular rules can be easily used to derive a type checking algorithm. Treating the first two positions of the relation as inputs and the last as an output leads directly to the definition of a recursive function for type-checking.

Now the type system can be used to restrict the set of valid programs to those that satisfy the judgment:

$$\bullet \vdash e : \tau$$

selecting those expressions that have some type with respect to the empty type environment, or are "well-typed". By making this restriction, we assert our belief that if a program is well-typed, then either it evaluates to a value or it does not terminate. Ideally, we should formally prove this property, but first it is helpful to test it. Modeling in Redex and testing properties such as this are the subject of the next section.

2.2. Modeling Semantics in Redex

The entire development of the previous section can be translated almost directly into Redex. In fact, all of the typesetting for the semantics of that section is generated automatically from a Redex model. In this section I present Redex's approach to semantics engineering by showing how it can be used to implement, inspect, and test such models.

Redex is an embedded domain-specific language. A domain-specific language (DSL) is one intended for a specific application, in this case semantics modeling. An embedded DSL is implemented as an extension to a general-purpose language (in Redex's case, Racket) instead of as a stand-alone tool. That enables the power of the general-purpose language to be used in combination with the targeted abstractions that the DSL provides. For Redex, this means that it is possible to "escape" to Racket when necessary, and all of tools and libraries already associated with Racket can be used in combination with Redex.

One of the core principles of Redex is to use already existing informal metalanguage found in programming language publications to guide its design. All of its core abstractions are chosen to model those programming language researchers have found to be commonly useful, such as grammars and reduction relations. Following this guideline makes designing useful abstractions simpler, as the choices have already been made by the community of intended users. It also has the potential to ease the learning curve of operational semantics.

```
(define-language STLC-min
  (e ::= (e e)
         (λ [x τ] e)
         (rec [x τ] e)
         (if0 e e e)
         (o e e)
         x n)
  (τ ::= (τ → τ)
         num)
  (n ::= number)
  (o ::= + -)
  (x ::= variable-not-otherwise-mentioned))
```

e ::= (e e)
  | (λ [x τ] e)
  | (rec [x τ] e)
  | (if0 e e e)
  | (o e e)
  | x | n

τ ::= ($\tau \to \tau$)
  | num

n ::= *number*

o ::= + | -

x ::= *variable-not-otherwise-mentioned*

Figure 4: Definition of a grammar in Redex (left) and the automatically generated typesetting.

A similar principle is applied to the design of Redex's syntax, which attempts to be as close as possible to what a semantics engineer would write down on the page or whiteboard. (Modulo some parentheses, the price of the embedding in Racket.) At the same time, automatic typesetting is provided that mimics what users see in the source as closely as possible, even preserving whitespace so that editing source code will directly affect typeset layouts, giving paper authors fine-grained control over layout.

A concrete example of Redex's approach is shown in figure 4, which compares the implementation of the core grammar from the previous section in Redex with the typeset version. The Redex form for defining a grammar is `define-language`, whose first argument is the name of the language, followed by a sequence of non-terminal definitions. Generating the typeset version on the right requires only a single line of code: `(render-language STLC-min)`. Note how the line-breaks and arrangement of productions on the right follow those in the source code. Finally, both the typeset version and the Redex source conform closely to commonly accepted ways of writing down a grammar. What is shown here is the raw automatic typesetting; Redex also provides hooks for customization, such as replacing *variable-not-otherwise-mentioned*, a special Redex pattern that matches anything that is not a literal in the language, with something more familiar. Similar correspondence between Redex source, Redex typesetting, and commonly accepted usage exists for all the Redex forms defining semantic elements.

After defining a language in Redex, it is straightforward to parse concrete syntax (in the form of s-expressions) according to the grammar. For example, the following interaction[3] uses the `redex-match` form to parse the term ((λ (*x* num) *x*) 5) as an application of one expression, `e_1`, in this language to another, `e_2`, where the `e`'s refer to the nonterminal of the language `STLC-min` from figure 4. The result is a representation of bindings from the patterns' two expressions to the relevant subterms for the one possible match in this case:

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

The `redex-match` syntactic form takes to a language defined as in figure 4, a patterned defined in reference to that language (in the above, for example the `e`'s refer to the non-terminal of the language), and a concrete term. It then attempts to parse to term according to the pattern. Trying to parse ((λ 4) 2), however, fails, since the first subterm no longer conforms to the *e* nonterminal, and is not a valid expression in this langauge:

```
> (redex-match STLC-min (e_1 e_2)
                        (term ((λ 4) 2)))
#f
```

Contexts, as introduced in section 2.1, are a native feature of patterns in Redex, and allow terms to be decomposed into a context with a hole and the content of the hole. For example:

[3]Inlined interactions that appear in this section are actual transcripts of the Racket REPL with the Redex module describing the language in the previous section loaded.

```
> (redex-match STLC (in-hole E n)
                    (term ((λ [x num] 6) 5)))
(list
 (match (list (bind 'E '((λ (x num) 6) hole)) (bind 'n 5))))
```

Here `in-hole` is Redex's notation for the application of a context, so `(in-hole E n)` is equivalent to $E[n]$ in the notation from section 2.1. (`STLC` is an extension of `STLC-min` that adds contexts.) The result tells us that there is exactly one way to decompose the term such that a number is in the hole, ((λ [x num] x) []) and 5. The `6` cannot appear in the hole, since in a function application with value on the left, the hole must be on the right.

Redex patterns also feature ellipses, which are analagous to the Kleene star and allow matching repetitions. A simple use case allows us to match a list of numbers of any length:

```
> (redex-match STLC (n ...)
                    (term (1 2 3 4 5)))
(list (match (list (bind 'n '(1 2 3 4 5)))))
```

A slightly more interesting example is to match a list of pairs of variables and numbers, a possible representation for an environment:

```
> (redex-match STLC ((x n) ...)
                    (term ((a 1) (b 2) (c 3) (d 4) (e 5))))
(list
 (match
  (list (bind 'n '(1 2 3 4 5)) (bind 'x '(a b c d e)))))
```

As a result we get back bindings for the variables `x`, a list of variables, and `n`, a list of numbers. Ellipses are a powerful feature of Redex's pattern matcher but cause problems for some types of random generation, an issue I will return to later on.

Reduction relations are defined using the `reduction-relation` form as a union of rules, the syntax of which is very close to that of figure 2. The definition of the reduction is shown on

```
(define STLC-red-one
  (reduction-relation
   STLC
   (--> ((λ [x τ] e) v)
        (subst e x v)
        β)
   (--> (rec [x τ] e)
        (subst e x (rec [x τ] e))
        μ)
   (--> (if0 0 e_1 e_2)
        e_1
        if-0)
   (--> (if0 n e_1 e_2)
        e_2
        (side-condition
         (term (different n 0)))
        if-n)
   (--> (o n_1 n_2)
        (δ n_1 o n_2)
        δ)))
```

```
(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 (x τ Γ) e τ)
   ----------------------
   (tc Γ (rec [x τ] e) τ)]
  [(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 (o e_0 e_1) num)])
```

Figure 5: Reduction-relation (left) and typing judgment definitions in Redex.

the left of figure 5. Each rule is parenthesized, and defined with the `-->` operator, which takes a left-hand-side pattern, and resulting term, a sequence of side conditions, and a rule name as its arguments.

To seen a reduction relation at work, we can use the `apply-reduction-relation` form, which takes a relation and a term to reduce one step:

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

A list containing one term is returned, since in this case there is only one possible reduction step, but depending on how the relation is defined, there could be more.

The typing judgment, shown on the right on figure 5, is also defined in a manner designed to follow the common syntax of figure 3. Instead of the designating the typing relation with the infix syntax $\Gamma \vdash e : \tau$, judgments in Redex code use parenthesized prefix-notation, in this case `(tc Γ e τ)`. Each rule is bracketed, and the conclusion appears below a horizontal line of dashes, the premises (and side-conditions) above. The only other significant addition is the mode annotation in the second line, which designates which positions of the relation are considered inputs and which are considered outputs. Redex requires this to ensure the judgment is executable without search, although it constrains the relations that can be expressed with `define-judgment-form`.

Judgments can be applied through the `judgment-holds` form. For example, we can verify that the type of `(+ 1 (- 2 3))` is a `num` as follows:

```
> (judgment-holds (tc • (+ 1 (- 2 3)) num))
#t
```

(Recall that • indicates the empty type environment.) Or, we can ask Redex to compute the type of a slightly more complicated term:

```
> (judgment-holds (tc • (λ [x num] (λ [y num] x)) τ) τ)
'((num → (num → num)))
```

And if we ask for the type of a term that is not well-typed, Redex returns `#f` to indicate the judgment does not hold:

```
> (judgment-holds (tc • (+ 7 (λ [y num] y)) τ) τ)
'()
```

Finally, to complete the Redex model of this language, we can define an `Eval` metafunction in Redex that corresponds exactly to Eval from section 2.1.

```
(define-metafunction STLC
  Eval : e -> n or function
  [(Eval e)
   n
   (judgment-holds (refl-trans e n))]
  [(Eval e)
   function
   (judgment-holds (refl-trans e (λ (x τ) e_3)))])
```

The first line specifies that this definition is relative to the STLC language and the second specifies Eval's contract. Two clauses follow, which are made up of, in order, a pattern, a result term, and a side-condition, which is where all the work of reducing the term is happening in this case. Clauses are tried in order, and the result is the right-hand side of the first clause that has both s pattern matching the argument and side-conditions that succeed. As before, the judgment-holds side-conditions in Eval apply the reflexive-transitive closure of the standard reduction (the judgment form refl-trans) to its argument and dispatch on the result. (Note that the side-conditions of the clauses differ in whether the result is an n or a λ-espression.) Metafunctions like Eval are applied as if they were functions in the object language, from within term. We can now evaluate programs using Redex. For example, to evaluate the application of the function that adds 1 to 1:

```
> (term (Eval ((λ [x num] (+ x 1)) 1)))
2
```

The unsurprising result is 2.

A more interesting example is:

```
(define (sumto n)
  (term ((rec [sumto (num → num)]
           (λ [x num]
             (if0 x
                  0
                  (+ x (sumto (- x 1))))))
```

```
            ,n)))
```

which defines a whole class of programs that calculate arithmetic series, sums of `1` through `n`. This definition takes advantage of Redex's status as an embedded language, defining a Racket function that returns a Redex `term`. The comma in the last line escapes to Racket, allowing the appropriate number to be inserted in the term.

Now we can try a slightly more interesting calculation, the value of the arithmetic series of `100`.

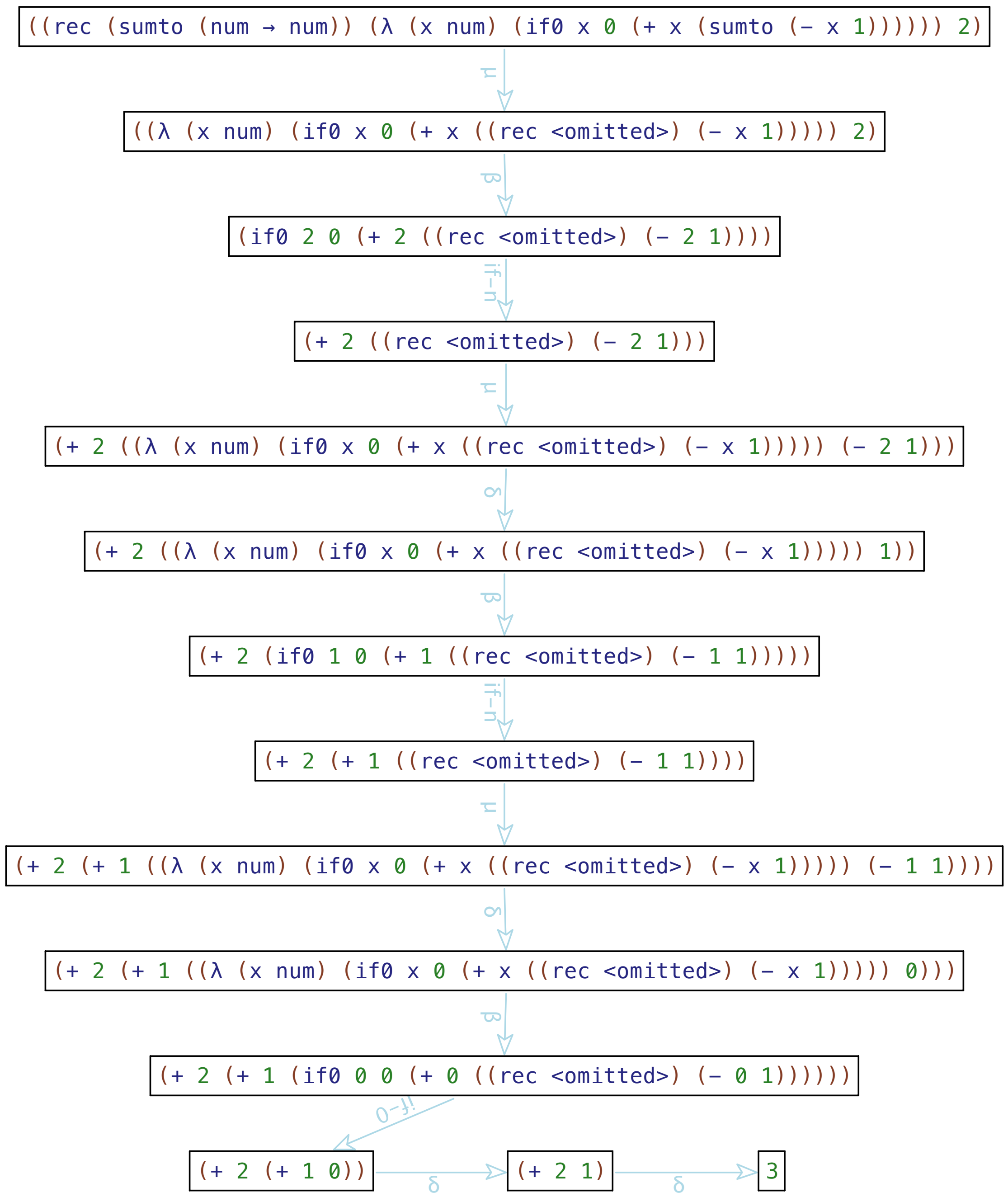
```
> (term (Eval ,(sumto 100)))
5050
```

which returns the answer we would expect.

Redex also allows us to observe the steps of a calculation with a reduction graph, where each two terms related by the reduction relation are nodes in the graph, and the edges are labeled with the rule that connects them. The above calculation actually has hundreds of steps, making its reduction graph too large for visual inspection. Figure 6, however, shows the reduction graph of an analagous program for the arithmetic series of `2`, showing each step from the initial program generated by `(sumto 2)` to the final value of `3`. Generating this reduction graph with Redex is again a one-liner given the appropriate definitions: `(traces STLC-red (sumto 2))`. The fact that there is a single path in the graph is a feature of this reduction relation; other reduction relations may give rise to many possible paths.

2.3. Property-based Testing with Redex

Redex strives to miminize the amount of time between sitting down to write a Redex model and having an exectuable semantics to work with. Executability alone already provides a significant



```
((rec (sumto (num → num)) (λ (x num) (if0 x 0 (+ x (sumto (- x 1)))))) 2)
((λ (x num) (if0 x 0 (+ x ((rec <omitted>) (- x 1))))) 2)
(if0 2 0 (+ 2 ((rec <omitted>) (- 2 1))))
(+ 2 ((rec <omitted>) (- 2 1)))
(+ 2 ((λ (x num) (if0 x 0 (+ x ((rec <omitted>) (- x 1))))) (- 2 1)))
(+ 2 ((λ (x num) (if0 x 0 (+ x ((rec <omitted>) (- x 1))))) 1))
(+ 2 (if0 1 0 (+ 1 ((rec <omitted>) (- 1 1)))))
(+ 2 (+ 1 ((rec <omitted>) (- 1 1))))
(+ 2 (+ 1 ((λ (x num) (if0 x 0 (+ x ((rec <omitted>) (- x 1))))) (- 1 1))))
(+ 2 (+ 1 ((λ (x num) (if0 x 0 (+ x ((rec <omitted>) (- x 1))))) 0)))
(+ 2 (+ 1 (if0 0 0 (+ 0 ((rec <omitted>) (- 0 1))))))
(+ 2 (+ 1 0))
(+ 2 1)
3
```

Figure 6: An example reduction graph. Since μ reductions substitute the entire rec expression in the body, the bodies of duplicate such expressions are omitted, but are all the same as the initial rec.

return on investment for the Redex user. Along with the tooling for interactive and graphical exploration of a model's dynamics, its greatest benefit is the ability to *test* a semantics.

An executable semantics can be equipped with a suite of tests. Unit tests alone, which Redex has built-in tooling for, provide a semantics engineer with the same benefits they are known to confer to software engineers, such as support for refactoring, the ability to apply test-driven development, and higher overall confidence in the correctness of a model. May Redex efforts are intended to model an already existing system (for example, Klein et al. (2013), Politz et al. (2013), Politz et al. (2012), and Guha et al. (2010)), and in such cases tests support confidence that the behaviors of the model and the implementation agree.

Unit tests, although unquestionably important for both software and semantics engineering, are fundamentally limited by the ingenuity of the writer of test cases. Another approach that relies less on human efforts to cover the space of possible tests is *property-based testing*, where instead of writing individual test cases a programmer defines a property that should hold of their program and a domain over which it should be checked. A *generator* then uses the definition of the domain to create many test cases in an attempt to falsify the property. QuickCheck (Claessen and Hughes 2000) and its many derivatives have popularized this approach over the past few years.

When developing a semantics we are usually interested in such general properties of a model, to the extent that a final step in such a development is to prove a number of them. For this reason, property-based testing is an especially attractive approach to semantics engineering. There are many testable properties that come up, and it is useful to be test them thoroughly during the development process.

Redex supports property-based testing through the `redex-check` form, which allows users to specify a method of generating terms and a property to check. It then generates a number of terms, checking the property with each one until it finds one that falsifies the property or reaches some maximum number of tries. As a first example, we can try to verify the (false) property that every expression in the language of section 2.2 is either a value or takes one step:

```
> (redex-check STLC e
               (or (redex-match STLC v (term e))
                   (not (empty?
                         (apply-reduction-relation STLC-red
                                                   (term e))))))
redex-check: counterexample found after 1 attempt:
a
```

The first line tells `redex-check` to generate random `e`'s from the `STLC` language, and the `or` expression is the predicate to check. In this case, it finds a counterexample on its first try, a free variable.

A property that should hold in this language is slightly more complex. If a term is well-typed, then it should be the case that it is either a value or that it can take a single reduction step. Also, if it can take a step, then the resulting term should have the same type. We can formulate this property as a Racket predicate:

```
(define (check-progress/preservation e)
  (define type-or-empty (judgment-holds (tc • ,e τ) τ))
  (define step-or-empty (apply-reduction-relation STLC-red e))
  (implies (not (empty? type-or-empty))
           (or (redex-match? STLC v e)
               (and (equal? 1 (length step-or-empty))
                    (equal? type-or-empty
                            (judgment-holds (tc • ,(car step-or-empty) τ)
                                            τ))))))
```

Now we can use `redex-check` to attempt to falsify it:

```
> (redex-check STLC e
               (check-progress/preservation (term e)))
```

```
redex-check: no counterexamples in 1000 attempts
```

This seems encouraging at first, but digging a little deeper exposes a common problem with random testing.

Test case generators can be called directly with the `generate-term` form, which take a language, a pattern, and a depth limit as parameters. This allows us to see what kind of terms `redex-check` is using to test the property:

```
> (generate-term STLC e 2)
'((if0 z 1 v) (rec (s num) Wr))
```

Clearly this term is not well-typed, it even has a number of free variables. Looking back at the definition of `check-progress/preservation` we can see that this isn't a very good test case, because it fails the premise of the implication that we want to test, so it doesn't verify that a well-typed term takes a step, or that the type is preserved. To check this part of the property, we need a good portion of the test cases to be well-typed. Also, we would like to avoid having too many of the well-typed test cases be values, because they won't take any evaluation steps. We can generate a number of terms with `generate-term` and check to see how many of them are "good":

```
> (length
   (filter (λ (e) (and (judgment-holds (tc • ,e τ))
                       (not (redex-match STLC v e))))
           (for/list ([i 1000])
             (generate-term STLC e 3))))
19
```

Here we generated 1000 random terms, and less than 2% of them are good test cases.

To give a better idea of what kind of terms are being generated, figure 7 shows some statistics for random terms in this language, and exposes some of the difficulty inherent in generating "good"

Term characteristic	**Percentage of Terms**
Reduces once or more	34.82%
Uses μ rule	29.96%
Well-typed	18.63%
Reduces twice or more	13.16%
Reduces three or more times	4.88%
Uses if-else rule	4.42%
Uses if-0 rule	4.28%
Well-typed, not a constant or constant function	2.45%
Uses β rule	2.31%
Uses δ rule	2.25%
Well-typed, reduces once or more	1.76%

Figure 7: Statistics for 100000 terms randomly generated from the stlc grammar.

terms. Although about 18% of random expresssions are well typed, only 2.45% are well-typed and not a constant or a constant function (a function of the form *(λ (x τ) n)*). The terms that are good tests for the property in question, those that are well-typed and exercise the reduction, are even rarer, at 1.76% of all terms.

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

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

Another approach is to use the type system directly to generate test cases. Adding this capability to Redex is one of the main contributions outlined in this dissertation. To do so, we can use the `#:satsifying` keyword with `generate-term`, which takes a judgment form that Redex will attempt to use to generate test cases.

```
> (generate-term STLC #:satisfying (tc • e τ) 2)
#f
```

We get back a complete type judgment containing a well-typed term and its type. Inspection of this term confirms that it is well-typed, and has no free variables.

Similarly, we can ask `redex-check` to use the typing judgment to generate its test cases.

```
> (redex-check STLC #:satisfying (tc • e τ)
               (check-progress/preservation (term e))
               #:attempts 100)
redex-check: no counterexamples in 100 attempts
```

Since all of the test cases used in this pass were well-typed, this is a much better test of the property, and provides higher confidence it is correct. Section 5 discusses how this type of generation works, and section 7 addresses how well different approaches to random generation do at testing semantics.

CHAPTER 3

Grammar-based Generators

A specification of abstract syntax is a fundamental part of any operational semantics. These specifications usually take the form of recursively defined data types or, as in Redex, regular tree grammars. These recursive structures are simple in the sense that they have no contextual constraints, unlike richer specifications such as type systems. Because of this they can be easily leveraged for a number of different approaches to random generation.

In this section I discuss two grammar-based approaches to random expression generation. First, I explain an approach based on recursively unfolding non-terminals, introduced to Redex by Klein and Findler (2009). Following that I address a newer method based on forming enumerations of the set of terms conforming to the grammar. Both approaches can be applied in general to any specification using abstract data types or regular grammars, although the discussion here is based on their implementation as part of Redex's random testing framework. The comparative effectiveness of the different approaches is discussed in section 7.

3.1. Ad-hoc Recursive Generators

Given a grammar, a straightforward method for generating random terms conforming to some non-terminal is as follows. First, pick a production at random. If that production does not include any non-terminals, we are done. If it contains non-terminals, we recur on them using the same method until reaching a non-recursive production.

To illustrate this approach, consider the following simple grammar for prefix-notation arithmetic expressions:

$$\begin{aligned} e &::= (o\ e\ e) \\ &\quad \mid n \\ o &::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \\ n &::= \textit{number} \end{aligned}$$

To generate a random expression in this language, we can start with the non-terminal *e* as our initial goal and transform it step-by-step into a term, where each step replaces a non-terminal with one of its productions:

$$e \longrightarrow (o\ e\ e) \longrightarrow (+\ e\ e) \longrightarrow (+\ n\ e) \longrightarrow (+\ 5\ e) \longrightarrow (+\ 5\ n) \longrightarrow (+\ 5\ 8)$$

Since in our grammar *number* is shorthand for the entire set of numbers, we just choose some element of that set for the non-terminal *n*.

It is straightforward to write a recursive function in Racket implementing this method. The only thing we need to be careful of is the danger of nontermination arising from randomly choosing recursive productions too often. We can deal with this by adding a "fuel" parameter that is decremented on recursive calls and only allows choosing recursive productions if it is positive, thus placing a limit on the depth of the generated term. Here is one implementation, as a function that takes a symbol indicating a non-terminal, a natural number indicating "fuel", and returns an appropriate random term:

```
(define/contract (generate-arith non-terminal fuel)
  (-> symbol? natural-number/c arith?)
  (define next-fuel (- fuel 1))
  (case non-terminal
    [(e)
     (define choice (if (> fuel 0) (random 2) 1))
     (if (= choice 0)
         (list (generate-arith 'o next-fuel)
               (generate-arith 'e next-fuel)
               (generate-arith 'e next-fuel))
```

```
        (generate-arith 'n next-fuel))]
   [(o)
    (list-ref '(+ - * /) (random 4))]
   [(n)
    (random 100)]))
```

(The predicate arith? appearing in the contract on the second line checks that the result does indeed conform to a non-terminal of the grammar above.) So, to generate a random expression, we can call generate-arith with 'e and a depth limit of 3, for a medium-sized term:

```
> (generate-arith 'e 3)
'(* (* 36 (+ 59 37)) 20)
```

The transformation from a grammar into such a function is easily automated. Redex's ad-hoc grammar generator performs just such a transformation, and although the grammars and patterns used in Redex can be significantly richer than those in this example, the method used is fundamentally the same.

Even in our simple example, however, we have made choices that can significantly affect the quality of our generator in testing. Most significantly, we have chosen to sample our numbers for the n non-terminal uniformly from integers in the interval between 0 and 100. This was done here for the sake of simplicity, and a little thought reveals it to be a very poor choice in general. A good test case generator should generate all types of numbers, which for Racket includes integers of unbounded magnitude, the usual floating point types, and even complex numbers. Further, a good generator should favor corner cases such as 0 and 1, and, in Racket's case, +inf.0, a number larger than all other numbers. Similar issues arise when generating strings or symbols. Neglecting this point is common in naive critiques of random test-case generators. Redex's grammar generator contains many such heuristics that have been tuned over years to make it more effective for random testing, which is why I refer to it as "ad-hoc." (See Klein and Findler (2009) for a discussion of some of these heuristics.)

Another concern about the testing effectiveness of generators of this type is that many properties that are desirable to test require preconditions that are much stronger than conformance to a grammar. For example, they may require closed terms, or even well-typed terms. In such cases the fraction of valid expressions conforming to a grammar that meet the stronger condition is usually very small. Even the ratio of closed lambda terms to lambda terms becomes vanishingly small as the size of terms increases, as shown by Grygiel and Lescanne (2013). To compensate, random generation from a grammar can still be leveraged by post-processing the term to fix these deficiencies. It is straightforward, for example, to write a function to eliminate free variables by adding new bindings or replacing them with closed subterms.

In spite of the issues with recursive grammar generation, it has been used many times over the years to great effect, starting with the landmark study of Hanford (1970). In fact, it is referred to as "the predominant generation strategy for language fuzzing" as recently as Dewey et al. (2014). It has been the default strategy in Redex since Klein and Findler (2009) and has been shown to be effective (when combined with a good post-processing function) in both a study testing the Racket virtual machine in Klein et al. (2011) and a case-study of models from ICFP 2009 conducted by Klein et al. (2012). Another recent tool based on this approach (although using sophisticated ad-hoc additions) is Csmith (Yang et al. 2011).

3.2. Grammar-based Enumerations

Another way of generating terms from a grammar is to construct an *enumeration* of the set of terms conforming to the grammar, a bijection between that set and the natural numbers. With an enumeration in hand, we can either generate terms in order or, if the enumeration is efficient, chose random natural numbers and decode them into the appropriate terms. Enumerations have been applied to property-based testing in a number of recent research efforts, notably Lazy Small Check (Runciman et al. 2008) and FEAT (Duregard et al. 2012). Here I discuss their application to grammars in Redex.

Redex enumerations are constructed using Racket's `data/enumerate` library (New 2014), which provides a rich set of combinators for constructing enumerations that are both efficient and fair. Efficiency means roughly that very large natural numbers can be decoded without too much computational cost. (In this case, it is usually linear in the size of the number in bits used to represent the index.) Fairness means that when combining different enumerations, such as when constructing an enumeration of an n-tuple out of n enumerations, the constituent enumerations are indexed into approximately evenly. Here I won't discuss the details of `data/enumerate`'s implementation, those are presented in New et al. (2015) along with a formal semantics of the library and a introduction to and proof of fairness. Instead I focus on its application in Redex, presenting enough of the API to support a description of how grammars are used to generate enumerations.

An enumeration in `data/enumerate` consists of a `to-nat` function that maps enumerated objects into the natural numbers, a `from-nat` function that maps the naturals into the enumerated objects, a size (the number of enumerated elements, possibly infinite), and a contract describing the enumerated elements. The simplest enumeration is the enumeration of natural numbers, where the bijection is just the identity. This is provided from `data/enumerate` as `natural/e`, and we can both decode and encode with it as follows:

```
> (to-nat natural/e 42)
42
> (from-nat natural/e 42)
42
```

Similarly, we can also construct enumerations of subsets of the naturals using `below/e`, which takes a number as its argument and returns a finite enumeration of the naturals up to less than the number.

Given some finite number of elements, we can construct an enumeration of them directly, using `fin/e`:

```
> (define abc/e (fin/e 'a 'b 'c 'd 'e 'f 'g))

> (to-nat abc/e 'c)
2
> (from-nat abc/e 5)
'f
```

Given two enumerations, we can combine them with `or/e`, which takes some number of enumerations as its arguments and returns their disjoint union. For example, we could form the combinations of the natural numbers and the enumeration above: `(or/e natural/e abc/e)`. The first 18 elements in that enumeration are:

```
0     'a    1     'b    2     'c    3     'd    4
'e    5     'f    6     'g    7     8     9     10
```

Note that we were able to combine finite and infinte enumerations in this example with no trouble, a necessary feature to build enumerations of Redex grammars.

Enumerations of tuples can be formed with `list/e`, which takes *n* enumerations as its arguments and returns the enumeration of the corresponding *n*-tuple. For example, the first 12 elements in the enumeration `(list/e natural/e natural/e natural/e)` are:

```
'(0 0 0)    '(0 0 1)    '(0 1 0)    '(0 1 1)
'(1 0 0)    '(1 1 0)    '(1 0 1)    '(1 1 1)
'(0 0 2)    '(1 0 2)    '(0 1 2)    '(1 1 2)
```

Only one more ingredient is necessary to be able to enumerate a simple grammar: `delay/e`, which enables the construction of fixed-points for recursively defined enumerations. To see how it works, we will build an enumeration for the same example grammar from the previous section:

$$\begin{aligned} e &::= (o\ e\ e) \\ &\quad\ \mid n \\ o &::= + \mid - \mid * \mid / \\ n &::= \textit{number} \end{aligned}$$

We can define an enumeration for this grammar as follows:

```
(define arith-e/e
  (letrec ([e/e (delay/e
                  (or/e n/e
                        (list/e o/e e/e e/e)))]
           [o/e (fin/e '+ '- '* '/)]
           [n/e (below/e 100)])
    e/e))
```

constructing an enumerator for each non-terminal in a mutually recursive manner. Enumerators for non-terminals that are self-recursive, such as `e/e`, are where `delay/e` is put to use, enabling the evaluation of the body to be delayed and unfolded as necessary.

Now we can construct the first few elements in the enumeration of the grammar:

```
0                  '(+ 0 0)           1
'(- 0 0)           2                  '(* 0 0)
3                  '(/ 0 0)           4
'(+ 0 (+ 0 0))     5                  '(- 0 (+ 0 0))
```

Or we can index more deeply into it:

```
> (from-nat arith-e/e 12345678987654321)
'(- (* (* 3 15) (/ 2 11)) (/ (* 3 15) (- 3 11)))
```

The efficiency of the enumeration combinators ensures that the above example completes almost instantaneously, as do even larger indices.

As with the ad-hoc grammar generator, given appropriate enumeration combinators, generating an enumeration from a grammar is for the most part straightforward. Each different pattern that can appear in a grammar definition is mapped into an enumeration. At a high-level, the correspondence between Redex patterns and the combinators is clear. Recursive non-terminals map into uses of `delay/e`, alternatives map into `or/e` and sequences map into `list/e`. Some care is also taken to exploit fairness. In particular, when enumerating the pattern, `(λ (x : τ) e)`, instead of generating list and pair patterns following the precise structure, which would lead to an unfair nesting, the pattern `(list/e x/e τ/e e/e)` is generated, where `x/e`, `τ/e` and `e/e` correspond to the enumerations for those non-terminals, from which the appropriate term is constructed.

CHAPTER 4

Derivation Generation by Example

This chapter introduces an alternative method for generating test-cases from a Redex program. In this approach, random derivations are constructed that satisfy judgment forms (and metafunctions) in a Redex model. In actual Redex models, this approach can be used to generate terms that are well-typed or satisfy some similar static property. Since such properties are frequently the premise of a testable property, that makes the terms useful as test cases.

Here I present an overview of the method for generating well-typed terms by working through the generation of an example term. (Section 5 provides a in-depth, formal explanation.) We will build a derivation satisfying the rules in figure 8, a subset of the rules for the typing judgment from the model in section 2.1. We begin with a goal pattern, which we will want the conclusion of the generated derivation to match.

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

$$\frac{(x\ \tau_x\ \Gamma) \vdash e : \tau_e}{\Gamma \vdash (\lambda\ [x\ \tau_x]\ e) : (\tau_x \rightarrow \tau_e)} \qquad \frac{\Gamma \vdash e_1 : (\tau_2 \rightarrow \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1\ e_2) : \tau}$$

Figure 8: Type system rules used in the example derivation.

Our goal pattern will be the following:

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

stating that we would like to generate an expression with arbitrary type in the empty type environment. We then randomly select one of the type rules. This time, the generator selects the abstraction rule, which requires us to specialize the values of e^0 and τ^0 in order to agree with the form of the rule's conclusion. To do that, we first generate a new set of variables to replace the ones in the abstraction rule, and then unify our conclusion with the specialized rule. We put a superscript 1 on these variables to indicate that they were introduced in the first step of the derivation building process, giving us this partial derivation.

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

The abstraction rule has added a new premise we must now satisfy, so we follow the same process with the premise. If the generator selects the abstraction rule again and then the application rule, we arrive at the following partial derivation, where the superscripts on the variables indicate the step where they were generated:

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

Application has two premises, so there are now two unfinished branches of the derivation. Working on the left side first, suppose the generator chooses the variable rule:

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

$$\begin{array}{lcl} \mathsf{lookup}[\![(x\ \tau\ \Gamma), x]\!] & = & \tau \\ \mathsf{lookup}[\![(x_1\ \tau\ \Gamma), x_2]\!] & = & \mathsf{lookup}[\![\Gamma, x_2]\!] \\ \mathsf{lookup}[\![\bullet, x]\!] & = & \#\mathrm{f} \end{array} \qquad \left| \qquad \begin{array}{c} \dfrac{}{\mathsf{lookup}[\![(x\ \tau\ \Gamma), x]\!] = \tau} \\[2ex] \dfrac{}{\mathsf{lookup}[\![\bullet, x]\!] = \#\mathrm{f}} \\[2ex] \dfrac{x_1 \neq x_2 \quad \mathsf{lookup}[\![\Gamma, x_2]\!] = \tau}{\mathsf{lookup}[\![(x_1\ \tau_x\ \Gamma), x_2]\!] = \tau} \end{array} \right.$$

Figure 9: Lookup as a metafunction (left), and the corresponding judgment form (right).

To continue, we need to use the lookup metafunction, whose definition is shown on the left-hand side of figure 9. Unlike judgment forms, however, Redex metafunction clauses are ordered, meaning that as soon as one of the left-hand sides matches an input, the corresponding right-hand side is used for the result. Accordingly, we cannot freely choose a clause of a metafunction without considering the previous clauses. Internally, our method treats a metafunction as a judgment form, however, adding premises to reflect the ordering.

For the lookup function, we can use the judgment form shown on the right of figure 9. The only additional premise appears in the bottom rule and ensures that we only recur with the tail of the environment when the head does not contain the variable we're looking for. The general process is more complex than lookup suggests and we return to this issue in section 5.2.

If we now choose that last rule, we have this partial derivation:

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

The generator now chooses lookup's first clause, which has no premises, thus completing the left branch.

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

Because pattern variables can appear in two different premises (for example the application rule's τ_2 appears in both premises), choices in one part of the tree affect the valid choices in other parts of the tree. In our example, we cannot satisfy the right branch of the derivation with the same choices we made on the left, since that would require $\tau_2^3 = (\tau_2^3 \rightarrow \tau^2)$.

This time, however, the generator picks the variable rule and then picks the first clause of the lookup, resulting in the complete derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{(x^2\ \tau_x^2\ (x^1\ (\tau_x^2 \rightarrow \tau^2)\ \bullet)) \vdash x^1 : (\tau_x^2 \rightarrow \tau^2)} \qquad \dfrac{\dfrac{}{\textsf{lookup}[\![(x^2\ \tau_x^2\ (x^1\ (\tau_x^2 \rightarrow \tau^2)\ \bullet)), x^2]\!] = \tau_x^2}}{(x^2\ \tau_x^2\ (x^1\ (\tau_x^2 \rightarrow \tau^2)\ \bullet)) \vdash x^2 : \tau_x^2}}{(x^2\ \tau_x^2\ (x^1\ (\tau_x^2 \rightarrow \tau^2)\ \bullet)) \vdash (x^1\ x^2) : \tau^2}}{(x^1\ (\tau_x^2 \rightarrow \tau^2)\ \bullet) \vdash (\lambda\ (x^2\ \tau_x^2)\ (x^1\ x^2)) : (\tau_x^2 \rightarrow \tau^2)}}{\bullet \vdash (\lambda\ (x^1\ (\tau_x^2 \rightarrow \tau^2))\ (\lambda\ (x^2\ \tau_x^2)\ (x^1\ x^2))) : ((\tau_x^2 \rightarrow \tau^2) \rightarrow (\tau_x^2 \rightarrow \tau^2))}$$

To finish the construction of a random well-typed term, we choose random values for the remaining, unconstrained variables, e.g.:

$$\bullet \vdash (\lambda\ (\textsf{f}\ (\textsf{num} \rightarrow \textsf{num}))\ (\lambda\ (\textsf{a}\ \textsf{num})\ (\textsf{f}\ \textsf{a}))) : ((\textsf{num} \rightarrow \textsf{num}) \rightarrow (\textsf{num} \rightarrow \textsf{num}))$$

We must be careful to obey the constraint that x^1 and x^2 are different, which was introduced earlier during the derivation, as otherwise we might not get a well-typed term. For example, $(\lambda\ (\textsf{f}\ (\textsf{num} \rightarrow \textsf{num}))\ (\lambda\ (\textsf{f}\ \textsf{num})\ (\textsf{f}\ \textsf{f})))$ is not well-typed but is an otherwise valid instantiation of the non-terminals.

CHAPTER 5

Derivation Generation in Detail

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

The grammar in figure 10 describes the language of the model. A program P consists of definitions D, which are sets of inference rules $((d\ p) \leftarrow a\ ...)$, here written horizontally with the conclusion on the left and premises on the right. Definitions can express both judgment forms and

$$\begin{aligned} P &::= (D\ ...) \\ D &::= (r\ ...) \\ r &::= ((d\ p) \leftarrow a\ ...) \\ a &::= (d\ p) \mid \delta \\ d &::= \mathit{Identifier} \end{aligned}$$

Programs

$$\begin{aligned} C &::= (\wedge\ (\wedge\ (x = p)\ ...) \\ &\qquad\ (\wedge\ \delta\ ...)) \\ e &::= (p = p) \\ \delta &::= (\forall\ (x\ ...)\ (\vee\ (p \neq p)\ ...)) \end{aligned}$$

Formulas

$$\begin{aligned} p &::= (\mathsf{lst}\ p\ ...) \\ &\mid m \\ &\mid x \\ m &::= \mathit{Constant} \\ x &::= \mathit{Variable} \end{aligned}$$

Patterns

Figure 10: The syntax of the derivation generator model.

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

Figure 11: Reduction rules describing generation of the complete tree of derivations.

metafunctions. They are a strict generalization of judgment forms, and metafunctions are compiled into them via a process we discuss in section 5.2.

The conclusion of each rule has the form $(d\ p)$, where d is an identifier naming the definition and p is a pattern. The premises a may consist of literal goals $(d\ p)$ or disequational constraints δ. We dive into the operational meaning behind disequational constraints later in this section, but as their form in figure 10 suggests, they are a disjunction of negated equations, in which the variables listed following $\forall$ are universally quantified. The remaining variables in a disequation are implicitly existentially quantified, as are the variables in equations.

The reduction relation shown in figure 11 generates the complete tree of derivations for the program P with an initial goal of the form $(d\ p)$, where d is the identifier of some definition in P and p is a pattern that matches the conclusion of all of the generated derivations. The relation is defined using two rules: [reduce] and [new constraint]. The states that the relation acts on are of the form $(P \vdash (a\ ...) \parallel C)$, where $(a\ ...)$ represents a stack of goals, which can either be incomplete derivations of the form $(d\ p)$, indicating a goal that must be satisfied to complete the derivation, or disequational constraints that must be satisfied. A constraint store C is a set of simplified equations and disequations that are guaranteed to be satisfiable. The notion of equality we use here is purely syntactic; two ground terms are equal to each other only if they are identical.

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

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

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

The remainder of this chapter fills in the details in this model and discusses the correspondence between the model and the implementation in more detail. First, an example Redex metafunction is translated into the model and used to generate a reduction graph in section 5.1. Section 5.2 describes the compilation of metafunctions, generalizing the process used for lookup in section 4.

$$\begin{array}{rl} n ::= & (\mathsf{s}\ n) \\ & \mid \mathsf{z} \\ b ::= & \mathsf{even} \mid \mathsf{odd} \end{array} \qquad\qquad \begin{array}{lcl} \mathsf{e/o}[\![\mathsf{z}]\!] & = & \mathsf{even} \\ \mathsf{e/o}[\![(\mathsf{s}\ (\mathsf{s}\ n))]\!] & = & \mathsf{e/o}[\![n]\!] \\ \mathsf{e/o}[\![n]\!] & = & \mathsf{odd} \end{array}$$

Figure 12: Grammar (left) and metafunction to be compiled and run in the derivation generator.

Section 5.3 describes how the solver handles equations and disequations. Section 5.4 discusses the heuristics in the implementation and section 5.5 describes how the implementation scales up to support features in Redex that are not covered in this model.

5.1. An Example

To get a better idea of how the model of the derivation generator works, this section works through the translation of a Redex metafunction into a program *P* of the model. Then, to see how the reduction in figure 11 works, a small but complete reduction graph is generated based on that program with a given initial goal.

Our starting point will be the simple grammar and metafunction e/o shown in figure 12. The language is a encoding of unary numbers *n*, and e/o is a function that takes an *n* and returns even or odd, depending on the number. It is written somewhat strangely to demonstrate some interesting aspects of the model.

As we did for the lookup metafunction in section 4, we can translate e/o into a judgment form defining a two-place relation by adding appropriate constraints as premises. The new judgment had the output of e/o (even or odd) in the first position and the corresponding unary number in the second. The rules for the judgment, which we'll call e-or-o, are shown in figure 13.

Since there is no overlap between the left-hand sides of the first two clauses in e/o the first two rules (reading left to right) have to additional premises. The third rule, however, has two premises

$$\frac{}{(\text{e-or-o } \mathsf{even}\ \mathsf{z})} \qquad \frac{(\text{e-or-o } b\ n)}{(\text{e-or-o } b\ (\mathsf{s}\ (\mathsf{s}\ n)))} \qquad \frac{(n \neq \mathsf{z}) \quad (\forall\ (n_1)\ (n \neq (\mathsf{s}\ (\mathsf{s}\ n_1))))}{(\text{e-or-o } \mathsf{odd}\ n)}$$

Figure 13: The metafunction of figure 12 as a judgment form.

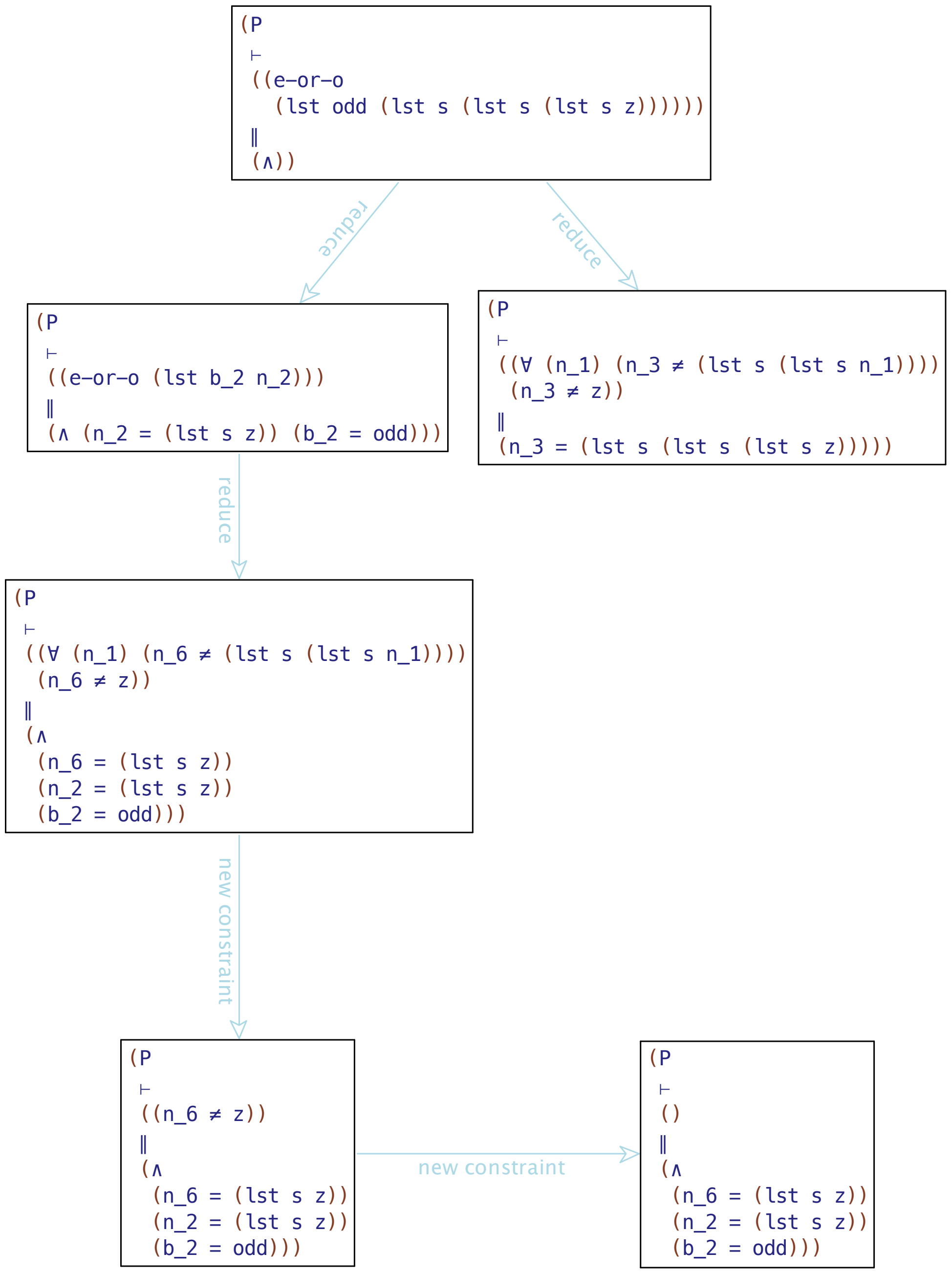
to exclude both of the previous clauses. The next section discusses in detail the form of such constraints and the need for universal quantification.

Finally, we would like to use e-or-o as a program for the derivation generator. We can directly translate the judgment of figure 13 into a definition *D* of the model as defined in figure 11. It is somewhat more difficult to read but is semantically identical:

```
(((e-or-o (lst even z)) ←)
 ((e-or-o (lst b (lst s (lst s n)))) ← (e-or-o (lst b n)))
 ((e-or-o (lst odd n))
  ←
  (∀ (n₁) (n ≠ (lst s (lst s n₁))))
  (n ≠ z)))
```

Note that sequence patterns all now have explicit lst constructors, corresponding to patterns *p* of the model, so that (s (s z)) becomes (lst s (lst s z)). Also, the parameters of the judgment have been combined into an lst sequence as well, since in the model all judgments are unary, so we just combine the parameters of any *n*-ary judgment into a tuple.

To generate a reduction graph for this program we need an appropriate initial goal, for which we can chose (e-or-o (list odd (lst s (lst s (list s z))))), asserting that three is odd, or that odd should be the result of calling e/o with 3. We then form a tuple from the program *P*, the goal, and the empty set of constraints (∧) (the constraint set is shown as a single disjunction for simplicity), which we use as an input for the reduction relation of figure 11. The resulting reduction graph is shown in figure 14.



```
(P
 ⊢
 ((e-or-o
   (lst odd (lst s (lst s (lst s z))))))
 ‖
 (∧))
```

```
(P
 ⊢
 ((e-or-o (lst b_2 n_2)))
 ‖
 (∧ (n_2 = (lst s z)) (b_2 = odd)))
```

```
(P
 ⊢
 ((∀ (n_1) (n_3 ≠ (lst s (lst s n_1))))
  (n_3 ≠ z))
 ‖
 (n_3 = (lst s (lst s (lst s z)))))
```

```
(P
 ⊢
 ((∀ (n_1) (n_6 ≠ (lst s (lst s n_1))))
  (n_6 ≠ z))
 ‖
 (∧
  (n_6 = (lst s z))
  (n_2 = (lst s z))
  (b_2 = odd)))
```

```
(P
 ⊢
 ((n_6 ≠ z))
 ‖
 (∧
  (n_6 = (lst s z))
  (n_2 = (lst s z))
  (b_2 = odd)))
```

```
(P
 ⊢
 ()
 ‖
 (∧
  (n_6 = (lst s z))
  (n_2 = (lst s z))
  (b_2 = odd)))
```

Figure 14: Reduction graph for example generator program. The program is abbreviated as P.

The first thing to notice about the reduction graph is that there are two possible reductions using the [reduce] rule from the initial state. Theses correspond to the middle and right rules from figure 13, both of which have conclusions that can be equated with the current goal. The right hand reduction comes from the rule on the right (note the form of the added constraints) and is a stuck state. Because it has a disequation δ on the top of the goal stack, it could only take a step using the [new constraint] rule. However, inspecting the current set of constraints (the bottom or last element of the state tuple) shows that it conflicts with the disequation at the top of the stack, so it isn't possible to take another step from this state.

The left hand reduction path takes a [reduce] step corresponding to a use of the middle rule from figure 13. It then takes another [reduce] step, but this time it can only use the rule on the right of the judgment, because `n_2` in the goal conflicts with both of the other possibilities, as it is defined by the constraint store to be `(lst s z)`. This step adds two new disequations, however neither of them conflict with the current constraint, and they are processed through two [new constraint] steps, resulting in a final state with an empty goal stack, representing a successful derivation.

No terms could be randomly generated with this reduction graph, since we started with a fully instantiated goal, and there is only one possible successful derivation. Starting with a different goal, such as one specifying `odd` in the first position and `n`, the pattern representing any possible unary number, in the second position would generate an infinitely branching reduction graph, with successful branches corresponding to each odd number, and stuck states in branches that would otherwise have led to an even number. The Redex implementation essentially executes a randomized search over such a reduction graph, looking for the successful branches.

5.2. Compiling Metafunctions

The examples of chapter 4 and section 5.1 informally demonstrated how metafunctions can be converted into judgment forms. This section discusses how to generalize this process.

The primary difference between a metafunction, as written in Redex, and a set of $((d\ p) \leftarrow a\ ...)$ clauses from figure 10 is sensitivity to the ordering of clauses. Specifically, when the second clause in a metafunction fires, then the pattern in the first clause must not match, in contrast to the rules in the model, which fire regardless of their relative order. Accordingly, the compilation process that translates metafunctions into the model must insert disequational constraints to capture the ordering of the cases.

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

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

The first clause matches any two-element list, and the second clause matches any pattern at all. Since the clauses apply in order, an application where the argument is a two-element list will reduce to 2 and an argument of any other form will reduce to 1. To generate conclusions of the judgment corresponding to the second clause, we have to be careful not to generate anything that matches the first.

Applying the same idea as lookup in section 4, we reach this incorrect translation:

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

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

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

$$\begin{aligned} P &::= (G \ldots) \\ G &::= D \mid M \\ M &::= (c \ldots) \\ p &::= \ldots. \\ &\quad \mid (f\, p) \\ c &::= ((f\, p) = p) \\ f &::= id \end{aligned}$$

Figure 15: Extensions to the language of figure 10 to add functions

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

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

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

To formalize this process as part of the model, we can first add (object-language) metafunctions to the language and then write (Redex) metafunctions to eliminate then. To ease confusion between the two language levels, I'll refer to object-language metafunctions as simply "functions," and the Redex equivalent as "metafunctions." The extensions adding functions to the language of the model are shown in figure 15. Programs become a list of either *D*'s or *M*'s, where *M* is a function definition as a list of clauses *c*. The name of the function *f* appears on the left-hand side of each clause, as in Redex. The other wrinkle is that patterns *p* are now allowed to include function applications (*f p*), reflecting that in Redex such applications are allowed inside `term`.

The metafunctions for function compilation are shown in figure 16. The top-level metafunction, compile, first calls compile-M with every function in the program until there are none, and then

$$\begin{aligned}
&\textsf{compile} : P \rightarrow (D \ldots)\\
&\textsf{compile}[\![(D \ldots)]\!] \quad = (\textsf{extract-apps-D}[\![D]\!] \ldots)\\
&\textsf{compile}[\![(D \ldots M\ G \ldots)]\!] = \textsf{compile}[\![(D \ldots \textsf{compile-M}[\![\textsf{freshen-cases}[\![M]\!]]\!]\ G \ldots)]\!]
\end{aligned}$$

$$\begin{aligned}
&\textsf{compile-M} : M \rightarrow D\\
&\textsf{compile-M}[\![(((f\ p_{in}) = p_{out}))]\!] =\\
&\quad (((f\ (\textsf{lst}\ p_{in}\ p_{out}))\ \leftarrow))\\
&\textsf{compile-M}[\![(((f_0\ p_1) = p_2)\ \ldots\ ((f\ p_{in}) = p_{out}))]\!] =\\
&\quad (r \ldots ((f\ (\textsf{lst}\ p_{in}\ p_{out}))\ \leftarrow (\forall\ \textsf{vars}[\![p_1]\!]\ (\vee\ (p_1 \neq p_{in})))\ \ldots))\\
&\ \text{where } (r \ldots) = \textsf{compile-M}[\![(((f_0\ p_1) = p_2)\ \ldots)]\!]
\end{aligned}$$

Figure 16: Metafunctions for compiling metafunctions *M* into definitions *D*.

calls extract-apps-D with every definition *D* in the program. The metafunction for compiling an individual function, compile-M, processes each prefix of the list of clauses individually, mapping each to a rule *r*. It does nothing special with the prefix containing only one clause, other than returning the equivalent rules, but in every other case it creates a rule based on the last clause in the prefix, adding in constraints excluding every other clause in the prefix.

Finally, the metafunctions shown in figure 17 lift out function applications embedded in patterns. An application of the form $(f\ p)$ is replaced with some fresh variable x representing its result. The application itself is transformed into a premise of the form $(f\ p\ x)$, since the function has been compiled into a relation of that form. The pattern p is in the input position and the variable x is in the output position. The premise is then lifted to the top level of the surrounding rule r. The extraction of all applications from within patterns and transformation to premises is performed by extract-apps-p, while the other metafunctions shown in figure 17 lift the premises to the rule level along with the appropriate bookkeeping.

The technique of converting functions into relations for use in a logic-programming setting is usually referred to as *flattening* and is not new, see for example Naish (1991) and Rouveirol (1994). However, the addition of disequations as part of the process has not previously been applied

$\textsf{extract-apps-D} : (r\ ...) \rightarrow (r\ ...)$
$\textsf{extract-apps-D}[\![(r\ ...)]\!] = (\textsf{extract-apps-r}[\![r]\!]\ ...)$

$\textsf{extract-apps-r} : r \rightarrow r$
$\textsf{extract-apps-r}[\![((d\ p) \leftarrow a\ ...)]\!] = ((d\ p_0) \leftarrow a_0\ ...\ (f_1\ p_1)\ ...\ (f_2\ p_2)\ ...\ ...)$
where $(p_0\ ((f_1\ p_1)\ ...)) = \textsf{extract-apps-p}[\![p]\!]$, $((a_0\ ((f_2\ p_2)\ ...))\ ...) = (\textsf{extract-apps-a}[\![a]\!]\ ...)$

$\textsf{extract-apps-a} : a \rightarrow (a\ (a\ ...))$
$\textsf{extract-apps-a}[\![(d\ p)]\!] = ((d\ p_0)\ ((f_1\ p_1)\ ...))$
where $(p_0\ ((f_1\ p_1)\ ...)) = \textsf{extract-apps-p}[\![p]\!]$
$\textsf{extract-apps-a}[\![(\forall\ (x\ ...)\ (\vee\ (p_1 \neq p_2)))]\!] = ((\forall\ (x\ ...)\ (\vee\ (p_1 \neq p_2)))\ ())$

$\textsf{extract-apps-p} : p \rightarrow (p\ (a\ ...))$
$\textsf{extract-apps-p}[\![(f\ p_0)]\!] = (x\ ((f\ (\textsf{lst}\ p\ x))\ (f_1\ p_1)\ ...))$
where $x = \textsf{fresh-var}[\![\textsf{y}]\!]$, $(p\ ((f_1\ p_1)\ ...)) = \textsf{extract-apps-p}[\![p_0]\!]$
$\textsf{extract-apps-p}[\![(\textsf{lst}\ p\ ...)]\!] = ((\textsf{lst}\ p_1\ ...)\ ((f_2\ p_2)\ ...\ ...))$
where $((p_1\ ((f_2\ p_2)\ ...))\ ...) = (\textsf{extract-apps-p}[\![p]\!]\ ...)$
$\textsf{extract-apps-p}[\![x]\!] = (x\ ())$
$\textsf{extract-apps-p}[\![m]\!] = (m\ ())$

Figure 17: Metafunctions for extracting function applications from within patterns.

and it allows a broader set of function definitions to be flattened, since otherwise, as noted by Naish (1991), one has to require that all left-hand sides be mutually non-unifiable. Further, and more importantly for Redex, it allows the rules of the resulting relation to be tried in any order. More recently, flattening has been applied to support test-case generation by Bulwahn (2013) for the purpose of inverting functions.

5.3. The Constraint Solver

The constraint solver maintains a set of equations and disequations that captures invariants of the current derivation that it is building. These constraints are called the constraint store and are kept in the canonical form C, as shown in figure 10, with the additional constraint that the equational portion of the store can be considered an idempotent substitution. That is, it always equates variables with with ps and, no variable on the left-hand side of an equality also appears in any right-hand side. Whenever a new constraint is added, consistency is checked again and the new set is simplified to maintain the canonical form.

Figure 18 shows solve, the entry point to the solver for new equational constraints. It accepts an equation and a constraint store and either returns a new constraint store that is equivalent to the conjunction of the constraint store and the equation or $\perp$, indicating that adding e is inconsistent with the constraint store. In its body, it first applies the equational portion of the constraint store as a substitution to the equation. Second, it performs syntactic unification (Baader and Snyder 2001) of the resulting equation with the equations from the original store to build a new equational portion of the constraint. Third, it calls check, which simplifies the disequational constraints and checks their consistency. Finally, if all that succeeds, check returns a constraint store that combines the results of unify and check. If either unify or check fails, then solve returns $\perp$.

Figure 19 shows dissolve, the disequational counterpart to solve. It applies the equational part of the constraint store as a substitution to the new disequation and then calls disunify. It disunify returns $\top$, then the disequation was already guaranteed in the current constraint store and thus does not need to be recorded. If disunify returns $\perp$ then the disequation is inconsistent with the current constraint store and thus dissolve itself returns $\perp$. In the final situation, disunify returns a new disequation, in which case dissolve adds that to the resulting constraint store.

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

The key trick in this function is to observe that since a disequation is always a disjunction of inequalities, its negation is a conjuction of equalities and is thus suitable as an input to unification. The first case in disunify covers the case where unification fails. In this situation we know that the

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

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

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

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

Figure 18: The Solver for Equations

disequation must have already been guaranteed to be false in constraint store (since the equational portion of the constraint store was applied as a substitution before calling disunify). Accordingly, disunify can simply return $\top$ to indicate that the disequation was redundant.

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

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

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

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

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

Figure 19: The Solver for Disequations

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

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

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

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

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

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

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

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

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

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

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

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

Figure 20: Metafunctions used to process disequational constaints.

This brings us to param-elim, in figure 20. Its first argument is a unifier, as produced by a call to unify to handle a disequation, and the second argument is the universally quantified variables from the original disequation. Its goal is to clean up the unifier by removing redundant and useless clauses.

There are two ways in which clauses can be false. In addition to clauses of the form $(x = p)$ where x is one of the universally quantified variables, it may also be the case that we have a clause of the form $(x_1 = x)$ and, as before, x is one of the universally quantified variables. This clause also must be dropped, according to the same reasoning (since = is symmetric). But, since variables on the right hand side of an equation may also appear elsewhere, some care must be taken here to avoid losing transitive inequalities. The function elim-x handles this situation, constructing a new set of clauses without *x* but, in the case that we also have ($x_2 = x$), adds back the equation ($x_1 = x_2$).

Finally, we return to check, shown in figure 20, which is passed the updated disequations after a new equation has been added in solve (see figure 18). It verifies the disequations and maintains their canonical form, once the new substitution has been applied. It does this by applying disunify to any non-canonical disequations.

Clearly, the soundness of the derivation generation process depends critically on the correctness of the constraint solver defined by solve and dissolve. Appendix A provides a formal proof of the correctness of the constraint solver. In addition, since it is comparatively inexpensive to do so, Redex's implementation checks that generated terms satisfy the judgment form or metafunction whose definition was used to generate them.

5.4. Search Heuristics

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

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

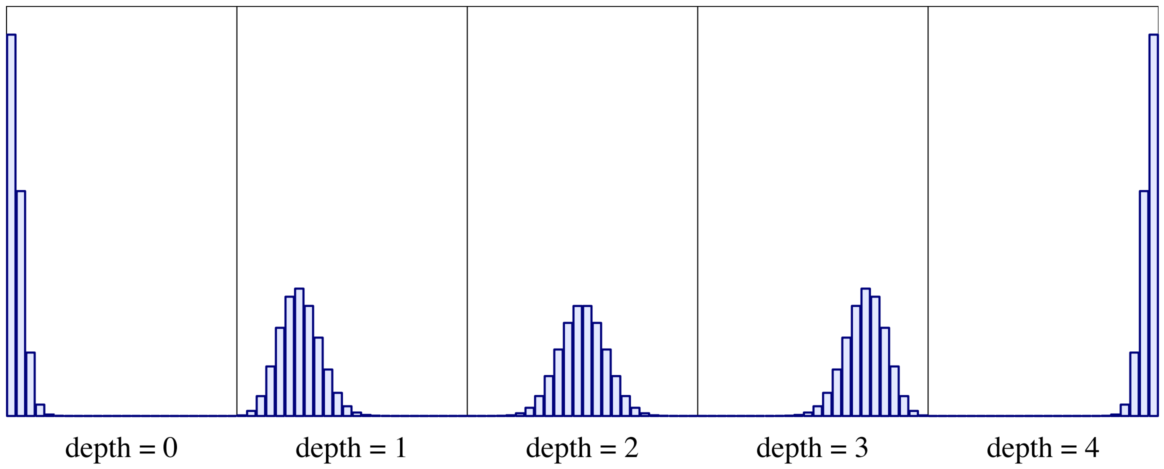


Figure 21: Density functions of the distributions used for the depth-dependent rule ordering, where the depth limit is 4 and there are 4 rules.

derivation, then this attempt is failure and the implementation either backtracks to an earlier such choice, or fails altogether.

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

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

More concretely, imagine that the depth bound was 4 and there are also 4 rules available. Accordingly, there are 24 different ways to order the premises. The graphs in figure 21 show the probability of choosing each permutation at each depth. Each graph has one x-coordinate for each different permutation and the height of each bar is the chance of choosing that permutation. The permutations along the x-axis are ordered lexicographically based on the number of premises that each rule has (so permutations that put rules with more premises near the beginning of the list are on the left and permutations that put rules with more premises near the end of the list are on the right). As the graph shows, rules with more premises are usually tried first at depth 0 and rules with fewer premises are usually tried first as the depth reaches the depth bound.

These two permutation strategies are complementary, each with its own drawbacks. Consider using the first strategy that gives all rule ordering equal probability with the rules shown in figure 8. At the initial step of our derivation, we have a 1 in 4 chance of choosing the type rule for numbers, so one quarter of all expressions generated will just be a number. This bias towards numbers also occurs when trying to satisfy premises of the other, more recursive clauses, so the distribution is skewed toward smaller derivations, which contradicts commonly held wisdom that bug finding is more effective when using larger terms. The other strategy avoids this problem, biasing the generation towards rules with more premises early on in the search and thus tending to produce larger terms. Unfortunately, experience testing Redex program suggests that it is not uncommon for there to be rules with large number of premises that are completely unsatisfiable when they are used as the first rule in a derivation (when this happens there are typically a few other, simpler rules that must be used first to populate an environment or a store before the interesting and complex

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

Figure 22: The subset of Redex's pattern language supported by the generator. Racket symbols are indicated by *s*, and *c* represents any Racket constant.

rule can succeed). For such models, using all rules with equal probability still is less than ideal, but is overall more likely to at least succeed.

Since neither strategy for ordering rules is always better than the other, Redex decides between the two randomly at the beginning of the search process for a single term, and uses the same strategy throughout that entire search. This is the approach the generator evaluated in section 7 uses.

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

5.5. A Richer Pattern Language

The model of section 5 uses a much simpler pattern language than Redex itself. The portion of Redex's internal pattern language supported by the generator is shown in figure 22. The generator is not currently able to handle parts of the pattern language that deal with evaluation contexts or "repeat" patterns (ellipses). This section discusses the interesting differences between this language and the language of the model, along with how they are supported in the implementation.

Named patterns of the form (name *s p*) correspond to variables *x* in the simplified version of the pattern language from figure 10, except that the variable *s* is paired with a pattern *p*. From the matcher's perspective, this form is intended to match a term with the pattern *p* and then bind the matched term to the name *s*. The generator pre-processes all patterns with a first pass that extracts the attached pattern *p* and attempts to update the current constraint store with the equation ($s = p$), after which *s* can be treated as a logic variable.

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

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

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

Patterns of the form (nt *s*) refer to a user-specified grammar, and match a term if it can be parsed as one of the productions of the non-terminal *s* of the grammar. It is less obvious how such non-terminal patterns should be dealt with in the unifier. To unify two such patterns, the intersection of two non-terminals should be computed, which reduces to the problem of computing the intersection of tree automata, for which there is no efficient algorithm (Comon et al. 2007). Instead a conservative check is used at the time of unification. When unifying a non-terminal with another pattern, an attempt is made to unify the pattern with each production of the non-terminal, replacing any embedded non-terminal references with the pattern any. We require that at least one of the unifications succeeds. Because this is not a complete check for pattern intersection, the names of the non-terminals are saved as extra information embedded in the constraint store until the entire generation process is complete. Then, once a concrete term is generated, it is checked to see if any of the non-terminals would have been violated (using a matching algorithm). This means that it is possible to get failures at this stage of generation, but it tends not to happen very often for practical Redex models. To be more precise, on the Redex benchmark (see Chapter 6) such failures occur on all “delim-cont” models 2.9$\pm$1.1% of the time, on all “poly-stlc” models 3.3$\pm$0.3% of the time, on the “rvm-6” model 8.6$\pm$2.9% of the time, and are not observed on the other models.

5.6. Related Work in Disequational Constraints

Colmerauer (1984) is the first to introduce a method of solving disequational constraints of the type used here, but his work handles only existentially quantified variables. Like him, Redex uses the unification algorithm to simplify disequations.

Comon and Lescanne (1989) address the more general problem of solving all first order logical formulas where equality is the only predicate, which they term “equational problems,” of which our constraints are a subset. They present a set of rules as rewrites on such formulas to transform them into solved forms. Redex's solver is essentially a way of factoring a stand-alone unifier out of their rules.

Byrd (2009) notes that a related form of disequality constraints has been available in many Prolog implementations and constraint programming systems since Prolog II. Notably, miniKanren (Byrd 2009) and cKanren (Alvis et al. 2011) implement them in a way similar to Redex, using unification as a subroutine. However, to my knowledge, none of these systems supports the universally quantified constraints Redex requires.

CHAPTER 6

The Redex Benchmark

This chapter introduces the Redex Benchmark, a suite of Redex models and bugs. The benchmark is intended to support the comparative evaluation of different methods of automated property-based testing. Section 6.1 discusses the problem of evaluating test-case generators and the approach used in this research, and section 6.2 describes the models and bugs used in the benchmark in detail. In section 7.1 the benchmark is applied to compare the different methods of random generation used by Redex.

6.1. Benchmark Rationale and Related Work

As Claessen and Hughes (2000) point out in the original paper on QuickCheck, it is "notoriously difficult" to evaluate the effectiveness of an approach to testing. Their paper provided strong anecdotal evidence that QuickCheck was effective for a variety of users with a variety of different applications, but didn't attempt a systematic study of its effectiveness. (A comparative study wasn't as easy to attempt at the time since their own tool was the first to popularize property-based testing for functional programmers.) In their case, the success of QuickCheck over the years has become the strongest evidence of its usefulness.

Subsequent efforts have made the attempt to be more systematic. In a study introducing SmallCheck, a property-based testing library for Haskell using exhaustive generation (as opposed to random generation in QuickCheck's case), Runciman et al. (2008) compare SmallCheck, Lazy SmallCheck, and QuickCheck on a few different programs: implementations of red-black trees, Huffman coding/decoding, a compiler from lambda calculus to combinators, and a chess problem solver. Since they are doing exhaustive testing, they give results for generation times of all inputs up to a certain depth, which one can argue should be correlated to testing effectiveness. In terms of finding actual faults, they report that two counterexamples were exposed during their study, both by Lazy SmallCheck, and give a cursory a description of one, for red black trees, where "a fault was fabricated in the rebalancing function by swapping two subtrees." The small number of counterexamples makes it difficult to draw solid conclusions about testing effectiveness.

A more recent comparative study was conducted by Bulwahn (2012) to evaluate a derivative of QuickCheck for the Isabelle (Nipkow et al. 2011) proof assistant. In this study, a number of different testing strategies were evaluated on a database of theorem mutations, faulty implementations of functional data structures, and an implementation of a hotel key-card system. The mutation database includes 400 mutated theorems from the areas of arithmetic, set theory, list data types, and examples drawn from the Isabelle Archive of Formal Proofs. The mutations were introduced by replacing constants and swapping arguments, and each testing method was given 30 seconds to find a counterexample to a given mutation. For the functional data structures, typos in the delete operation were introduced to create faulty versions of AVL, red-black, and 2-3 trees, and the property that delete preserved balance and ordering was tested, again with a time limit of 30 seconds. The hotel key-card system allowed a possible man-in-the-middle attack, which one method was able to find in ten minutes.

The Redex benchmark attempts as much as possible to measure how effective different automated testing methods are at finding counterexamples to real-world bugs on real-world models. Models come from two sources: pre-existing Redex models and those synthesized for the benchmark. In both cases an effort has been made to use models that are "typical" of those that Redex users write. Bugs are inserted by hand into the models, and are either actual bugs introduced and found during the development of the model, or inserted because they are judged to be representative of bugs that could be introduced in a typical development. A short description of each model and each bug is given in section 6.2.

The models themselves represent a wider variety and a deeper complexity than those used in previous studies. As in both studies mentioned above, we include an implementation of a functional data structure, namely red-black trees. The rest of the models, however, are programming languages or virtual machines that typically have much richer properties to test, such as type-soundness. This provides a broader range of models and properties to test and targets the domain (PL semantics) for which Redex's automated testing support is intended.

We also test the models for much longer time periods, up to 24 hours (or more, if uncertainty remains large) for each generator/bug pair. This is intended to coincide more closely with actual use cases, where a test run may frequently extend over lunch, overnight, or a weekend. It also exposes differences at larger time scales that can be exploited through optimization of the testing method or parallelism. (Since test runs are independent, it is easy to take advantage of parallelism in this setting.)

Finally, as a metric we choose the (average) time to find a counterexample. This measures exactly the property we desire in a test generator. Other possibilities, such as the time to exhaust a finite space of possible test cases, or the ratio of attempts to counterexample, are also interesting, but are not as general. A smaller number of attempts per counterexample, for example, may be desirable, but not if the cost per attempt becomes too large. We regard such more specific properties as useful in diagnosing or improving the performance of a specific generator, but not for making the type of general comparisons we are interested in examining with the benchmark.

Model	synthesized	artifact	loc	# of bugs
delim-cont		●	287	3
let-poly	●		640	7
list-machine		●	256	3
poly-stlc	●	●	277	9
rbtrees	●		187	3
rvm		●	712	7
stlc	●		211	9
stlc-sub	●		241	9

Figure 23: Benchmark Models

6.2. The Benchmark Models

The programs in the benchmark come from two sources: synthetic examples based on experience with Redex over the years and from pre-existing models along with bugs that were encountered during the development process.

The benchmark has six different Redex models, each of which provides a grammar of terms for the model and a soundness property that is universally quantified over those terms. Most of the models are of programming languages and most of the soundness properties are type-soundness, but we also include red-black trees with the property that insertion preserves the red-black invariant, as well as one richer property for one of the programming language models (discussed in section 6.2.3). Figure 23 summarizes the models included, showing whether they are synthesized for the benchmark or a pre-existing artifact that was included, the non-whitespace, non-comment lines of code, and the number of bugs added to each model. The line number counts include the model and the specification of the property.

For each model, bugs are manually introduced into a number of copies of the model, such that each copy is identical to the correct one, except for a single bug. The bugs always manifest as a term that falsifies the soundness property.

Each bug has been classified by hand according to a qualitative scheme as **S/M/D/U**, meaning, as follows:

- **S** (Shallow) Errors in the encoding of the system into Redex, due to typos or a misunderstanding of subtleties of Redex.
- **M** (Medium) Errors in the algorithm behind the system, such as using too simple of a data-structure that doesn't allow some important distinction, or misunderstanding that some rule should have a side-condition that limits its applicability.
- **D** (Deep) Errors in the developer's understanding of the system, such as when a type system really isn't sound and the author doesn't realize it.
- **U** (Unnatural) Errors that are unlikely to have come up in real Redex programs but are included for our own curiosity. There are only two bugs in this category.

The table in Appendix B gives a more detailed overview, showing the classification, size of the smallest known counterexample, and a short description for each bug. Each bug has a number and, with the exception of the rvm model, the numbers count from 1 up to the number of bugs. The rvm model bugs are all from Klein et al. (2013)'s work and we follow their numbering scheme (see section 6.2.8 for more information about how we chose the bugs from that paper).

The following subsections each describe one of the models in the benchmark, along with the errors introduced into each model. The bugs are described along with short justifications for how they are categorized.

6.2.1. stlc

A simply-typed λ-calculus with base types of numbers and lists of numbers, including the constants `+`, which operates on numbers, and `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 preservation (if a term has a type and takes a step, then the resulting term has the same type) and progress (that well-typed non-values always take a reduction step).

Nine different bugs wre introduced into this system. The first confuses the range and domain types of the function in the application rule, and has the small counterexample: `(hd 0)`. We consider this to be a shallow bug, since it is essentially a typo and it is hard to imagine anyone with any knowledge of type systems making this conceptual mistake. Bug 2 neglects to specify that a fully applied `cons` is a value, thus the list `((cons 0) nil)` violates the progress property. We consider this be be a medium bug, as it is not a typo, but an oversight in the design of a system that is otherwise correct in its approach.

We consider the next three bugs to be shallow. Bug 3 reverses the range and the domain of function types in the type judgment for applications. Bug 4 assigns `cons` a result type of `int`. The fifth bug returns the head of a list when `tl` is applied. Bug 6 only applies the `hd` constant to a partially constructed list (i.e., the term `(cons 0)` instead of `((cons 0) nil)`).

The seventh bug, also classified as medium, omits a production from the definition of evaluation contexts and thus doesn't reduce the right-hand-side of function applications.

Bug 8 always returns the type `int` when looking up a variable's type in the context. This bug (and the identical one in the next system) are the only bugs we classify as unnatural. It is included because it requires a program to have a variable with a type that is more complex that just `int` and to actually use that variable somehow.

Bug 9 is simple; the variable lookup function has an error where it doesn't actually compare its input to variable in the environment, so it effectively means that each variable has the type of the nearest enclosing lambda expression.

6.2.2. poly-stlc

This is a polymorphic version of the model in section 6.2.1, with a single numeric base type, polymorphic lists, and polymorphic versions of the list constants. No changes were made to the model except those necessary to make the list operations polymorphic. There is no type inference in the model, so all polymorphic terms are required to be instantiated with the correct types in order to type check. Of course, this makes it much more difficult to automatically generate well-typed terms, and thus counterexamples. As with **stlc**, the property checked is type soundness.

All of the bugs in this system are identical to those in **stlc**, aside from any changes that had to be made to translate them to this model.

This model is also a subset of the language specified in Pałka et al. (2011), who used a specialized and optimized QuickCheck generator for a similar type system to find bugs in GHC. This system as adapted (along with its restriction in **stlc**) because it has already been used successfully with random testing, which makes it a reasonable target for an automated testing benchmark.

6.2.3. stlc-sub

This is the same language and type system as section 6.2.1, except that in this case all of the errors are in the substitution function.

Experience with Redex shows it is easy to make subtle errors when writing substitution functions, and this set of tests specifically targets them with the benchmark. There are two soundness checks for this system. Bugs 1-5 are checked in the following way: given a candidate counterexample, if it type checks, then all βv-redexes in the term are reduced (but not any new ones that might appear) using the buggy substitution function to get a second term. Then, these two terms are checked to see if they both still type check and have the same type and that the result of passing both to the evaluator is the same.

Bugs 4-9 are checked using type soundness for this system as specified in the discussion of the section 6.2.1 model. We included two predicates for this system because we believe the first to be a good test for a substitution function but not something that a typical Redex user would write, while the second is something one would see in most Redex models but is less effective at catching bugs in the substitution function.

The first substitution bug introduced simply omits the case that replaces the correct variable with the term to be substituted. We consider this to be a shallow error. Bug 2 permutes the order of arguments when making a recursive call. This is also categorized as a shallow bug, although it is a common one, at least based on experience writing substitutions in Redex. Bug 3 swaps the function and argument positions of an application while recurring, again essentially a typo and a shallow error, although one of the more difficult to find in this model.

The fourth substitution bug neglects to make the renamed bound variable fresh enough when recurring past a lambda. Specifically, it ensures that the new variable is not one that appears in the body of the function, but it fails to make sure that the variable is different from the bound variable or the substituted variable. We categorized this error as deep because it corresponds to a misunderstanding of how to generate fresh variables, a central concern of the substitution function. Bug 5 carries out the substitution for all variables in the term, not just the given variable. We categorized it as SM, since it is essentially a missing side condition, although a fairly egregious one. Bugs 6-9 are duplicates of bugs 1-3 and bug 5, except that they are tested with type soundness instead. (It is impossible to detect bug 4 with this property.)

6.2.4. let-poly

A language with ML-style `let` polymorphism, included in the benchmark to explore the difficulty of finding the classic let+references unsoundness. With the exception of the classic bug, all of the bugs were errors made during the development of this model (and that were caught during development).

The first bug is simple; it corresponds to a typo, swapping an `x` for a `y` in a rule such that a type variable is used as a program variable. Bug number 2 is the classic let+references bug. It changes the rule for `let`-bound variables in such a way that generalization is allowed even when the initial value expression is not a value. This is a deep bug. Bug number 3 is an error in the function application case where the wrong types are used for the function position (swapping two types in the rule). Bugs 4, 5, and 6 were errors in the definition of the unification function that led to various bad behaviors. Bug 4 is a simple typo, while 5 and 6 are actual errors although not deep ones, and are classified as medium.

Finally, bug 7 is a bug that was introduced early on, but was only caught late in the development process of the model. It used a rewriting rule for `let` expressions that simply reduced them to the corresponding `((`λ expressions. This has the correct semantics for evaluation, but the statement of type-soundness does not work with this rewriting rule because the let expression has more polymorphism that the corresponding application expression, a subtle point that is easy to get wrong, so this was classified as a deep bug.

6.2.5. list-machine

An implementation of Appel et al. (2012)'s list-machine benchmark. 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. (2012), namely that well-typed programs always step or halt. Three mutations are included.

The first list-machine bug incorrectly uses the head position of a cons pair where it should use the tail position in the cons typing rule. This bug amounts to a typo and is classified as simple.

The second bug is a missing side-condition in the rule that updates the store that has the effect of updating the first position in the store instead of the proper position in the store for all of the store update operations. We classify this as a medium bug.

The final list-machine bug is a missing subscript in one rule that has the effect that the list cons operator does not store its result. Essentially a typo, and classified as a simple bug.

6.2.6. rbtrees

A model that implements the red-black tree insertion function and checks that insertion preserves the red-black tree invariant (and that the red-black tree is a binary search tree).

The first bug simply removes the re-balancing operation from insert. We classified this bug as medium since it seems like the kind of mistake that a developer might make in staging the implementation. That is, the re-balancing operation is separate and so might be put off initially, but then forgotten.

The second bug misses one situation in the re-balancing operation, namely when a black node has two red nodes under it, with the second red node to the right of the first. This is a medium bug.

The third bug is in the function that counts the black depth in the red-black tree predicate. It forgets to increment the count in one situation. As a small oversight, this is a simple bug.

6.2.7. delim-cont

Takikawa et al. (2013)'s model of a contract and type system for delimited control. The language is Plotkin's PCF extended with operators for delimited continuations, continuation marks, and contracts for those operations. The property checked is type soundness. We added three bugs to this model.

The first was a bug found by mining the model's git repository's history. This bug fails to put a list contract around the result of extracting the marks from a continuation, which has the effect of checking the contract that is supposed to be on the elements of a list against the list itself instead. We classify this as a medium bug.

The second bug was in the rule for handling list contracts. When checking a contract against a cons pair, the rule didn't specify that it should apply only when the contract is actually a list contract, meaning that the cons rule would be used even on non-list contacts, leading to strange contract checking. We consider this a medium bug because the bug manifests itself as a missing `list/c` in the rule.

The last bug in this model makes a mistake in the typing rule for the continuation operator. The mistake is to leave off one-level of arrows, something that is easy to do with so many nested arrow types, as continuations tend to have. We classify this as a simple error.

6.2.8. rvm

A existing model and test framework for the Racket virtual machine and bytecode verifier (Klein et al. 2013). The bugs were discovered during the development of the model and reported in section 7 of that paper. Unlike the rest of the models, bugs for this model are not numbered sequentially but instead use the numbers from Klein et al. (2013)'s work. The bugs are described in detail in Klein et al. (2013)'s paper.

Only some bugs from the paper were used, excluding bugs for two reasons:

- The paper tests two properties: an internal soundness property that relates the verifier to the virtual machine model, and an external property that relates the verifier model to the verifier implementation. Those that require the latter properties were excluded because it requires building a complete, buggy version of the Racket runtime system to include in the benchmark.
- All of the internal properties were included, except those numbered 1 and 7, for practical reasons. The first is the only bug in the machine model, as opposed to just the verifier, which would have required the inclusion of the entire VM model in the benchmark. The second would have required modifying the abstract representation of the stack in the verifier model in contorted way to mimic a more C-like implementation of a global, imperative stack. This bug was originally in the C implementation of the verifier (not the Redex model) and to replicate it in the Redex-based verifier model would require programming in a low-level imperative way in the Redex model, something not easily done.

This model is unique in our benchmark suite because it includes a function that makes terms more likely to be useful test cases. In more detail, the machine model does not have variables, but instead is stack-based; bytecode expressions also contain internal pointers that must be valid. Generating a random (or in-order) term is relatively unlikely to produce one that satisfies these constraints. For example, of the first 10,000 terms produced by the in-order enumeration only 1625 satisfy the constraints. The ad hoc random generator generators produces about 900 good terms in 10,000 attempts and the uniform random generator produces about 600 in 10,000 attempts.

To make terms more likely to be good test cases, this model includes a function that looks for out-of-bounds stack offsets and bogus internal pointers and replaces them with random good values. This function is applied to each of the generated terms before using them to test the model.

CHAPTER 7

Evaluation

This chapter reports on two studies. In the first, all of the generators described in this dissertation are evaluated using the Redex benchmark. The second study compares the derivation generator of section 5 to a similar generator that is the best-known hand-tuned typed term generator.

7.1. The Redex Benchmark

This section details a comparison of all of Redex's approaches to generation on the Redex benchmark. This includes the the three grammar-based generators of section 3 (the ad-hoc recursive generator, in-order enumeration, and random selection from an enumeration) and the derivation generator of section 5.

The generators were compared using the Redex Benchmark of chapter 6. For a single test run, we pair a generator with a model and its soundness property, and then repeatedly generate test cases using the generator, testing them with the soundness property. We track the intervals between instances where the test case causes the soundness property to fail. For this study, each run continued for either 24 hours or until the uncertainty in the average interval between such counterexamples became acceptably small.

The enumerations described in section 3.2 were used to build two generators, one that just chooses terms in the order induced by the natural numbers (referred to below as in-order), and one that selects a random natural and uses that to index into the enumeration (referred to as random idnexing).

To pick a random natural number to index into the enumeration, first an exponent i in base 2 is chosen from the geometric distribution and then an integer that is between 2^{i-1} and 2^i is picked uniformly at random. This process is repeated three times and then the largest is chosen, which helps make sure that the numbers are not always small. This distribution is used because it does not have a fixed mean. That is, if you take the mean of some number of samples and then add more samples and take the mean again, the mean of the new numbers is likely to be larger than from the mean of the old. This is a good property to have when indexing into our enumerations to avoid biasing indices towards a small size.

The random indexing results are sensitive to the probability of picking the zero exponent from the geometric distribution. Because this method is the worst performing method, benchmark-specific numbers were empirically chosen in an attempt to maximize the success of the random enumeration method. Even with this artificial help, this method was still worse, overall, than the other three.

All of the other generators except in-order enumeration have some parameter controlling the maximum size of generated terms.

The ad-hoc random generator, which is based on the method of recursively unfolding non-terminals, is parameterized over the depth at which it attempts to stop unfolding non-terminals. A value of 5 was chosen for this depth since that seemed to be the most successful. This produces terms of a similar size to those of the random enumeration method, although the distribution is different.

The derivation generator is similarly parameterized over the depth at which it attempts to begin finishing off the derivation, or where it begins to prefer less-recursive premises. Values that produced terms of a similar size to the ad-hoc generator were chosen, except in cases where this caused too many search failures, in which case a smaller depth was used. The depths used range from 3 to 5.

Each generator was applied to all of the bugs in the benchmark, with the exception of the derivation generator, which isn't able to handle the **let-poly** model. For reasons that have to do with the way Redex handles variable freshness, the typing judgment for this model is written using an explicit continuation and all recursive judgments have only one premise. Because of this, the heuristics that the derivation generator uses fail when applied to this model. This causes a runaway search process that is eventually terminated by constraints Redex imposes and deemed a failure. Thus there are 7 bugs that the derivation generator could not be tested on.

There are 50 bugs total in the benchmark, for a total of 193 bug/generator pairs in this study. For each of the bug and generator combinations, a script is run that repeatedly asks for terms and checks to see if they falsify the model's correctness property. As soon as it finds a counterexample to the property, it reports the amount of time it has been running. The script was run in two rounds. The first round ran all 193 bug and generator combinations until either 24 hours elapsed or the standard error in the average became less than 10% of the average. Then all of the bugs where the 95% confidence interval was greater than 50% of the average and where at least one counterexample was found were run for an additional 8 days. All of the final averages have an 95% confidence interval that is less than 50% of the average.

Two identical 64 core AMD machines with Opteron 6274s running at 2,200 MHz with a 2 MB L2 were used cache to run the benchmarks. Each machine has 64 gigabytes of memory. The script typically runs each model/bug combination sequentially, although multiple different combinations are run in parallel and, for the bugs that ran for more than 24 hours, tests are in parallel. We used version 6.2.900.4 (from git on August 15, 2015) of Racket, of which Redex is a part.

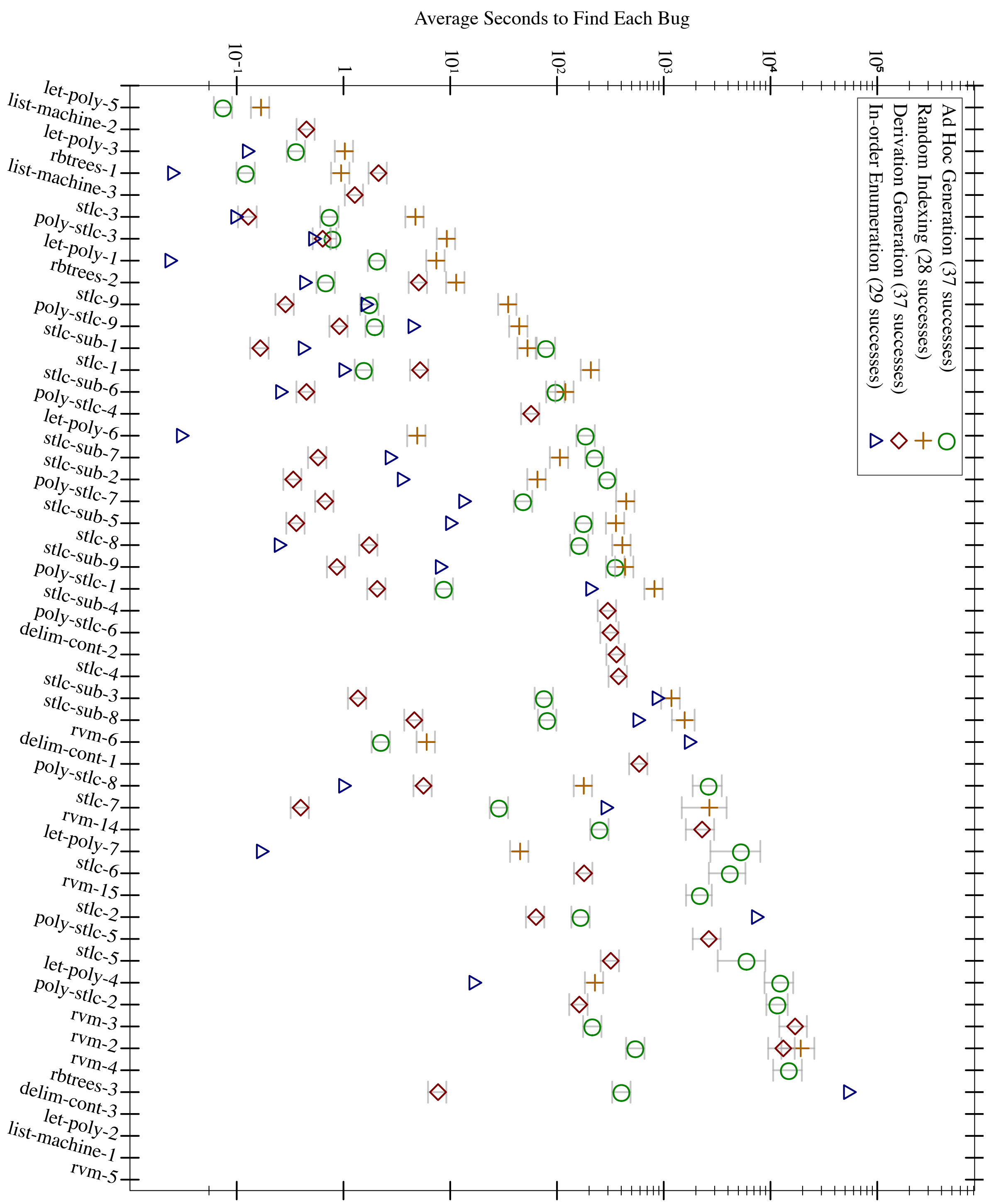


Figure 24: Benchmark results for all generators on all bugs. Error bars show 95% confidence intervals.

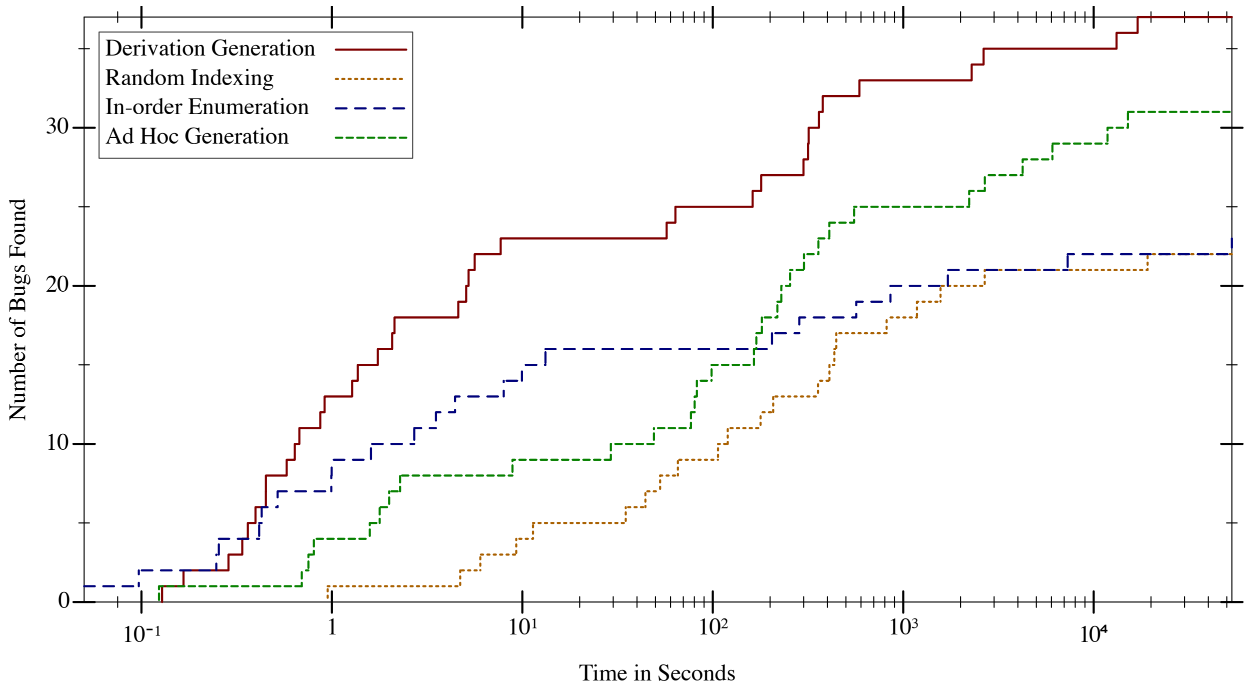


Figure 25: Random testing performance of all four generators, on models where all generators apply.

Figure 24 summarizes the results of the comparison on a per-bug basis. The y-axis is time in seconds, and for each bug the average time it took each generator to find a counterexample is plotted. The bugs are arranged along the x-axis, sorted by the average time over all generators to find the bug. The error bars represent 95% confidence intervals in the average, and in all cases where the averages differ significantly the errors are small enough to clearly differentiate the averages. The three blank columns on the right are bugs that no generator was able to find. The vertical scale is logarithmic, and the average time ranges from a tenth of a second to several hours, an extremely wide range in the rarity of counterexamples.

To depict more clearly the relative testing effectiveness of the generation methods, the same data is plotted slightly differently in figure 25. Here the time in seconds is shown on the x-axis (the y-axis from figure 24, again on a log scale), and the total number of bugs found for each point in time on the y-axis. This plot only includes bugs to which all generators can be applied, to avoid having this aspect of the benchmark's composition unduly affect the comparison. (Therefore, **let-poly** is excluded since the derivation generator cannot handle it.) This plot makes it clear that the derivation generator is much more effective when it applies, finding more bugs more quickly at almost every time scale. In fact, an order of magnitude or more on the time scale separates it and the next-best generator for almost the entire plot.

While the derivation generator is more effective when it is used, it cannot be used with every Redex model, unlike the other generators. There are three broad categories of models to which it may not apply. First, the language may not have a type system, or the type system's implementation might use constructs that the generator fundamentally cannot handle (like escaping to Racket code to run arbitrary computation). Second, the generator currently cannot handle ellipses (aka repetition or Kleene star). And finally, some judgment forms thwart its termination heuristics. Specifically, the heuristics make the assumptions that the cost of completing the derivation is proportional to the size of the goal stack, and that terminal nodes in the search space are uniformly distributed. Typically these are safe assumptions, but not always; as noted already, the **let-poly** model breaks them.

Figure 26 shows the testing performance on all bugs in the benchmark for the generators that are able to attempt all of them. This reveals that the ad hoc generator is better than the best enumeration strategy after 22 minutes. Before that time, the in-order enumeration strategy is the best approach, and often by a significant margin. Random indexing into an enumeration is never the best strategy.

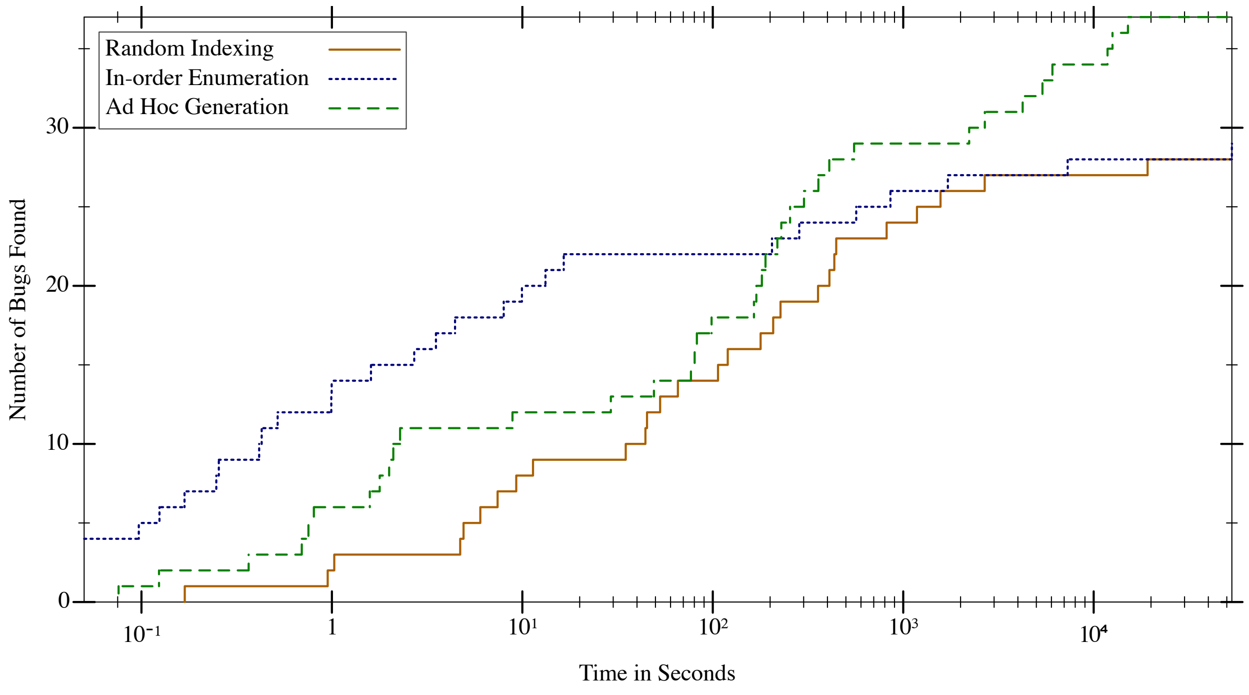


Figure 26: Random testing performance of ad-hoc and enumeration generators on all models.

7.2. Testing GHC: A Comparison With a Specialized Generator

In this section, the derivation generator developed in this work for Redex is compared to a specialized generator of typed terms. The specialized generator was designed to be used for differential testing of GHC, and generates terms for a specific variant of the lambda calculus with polymorphic constants, chosen to be close to the compiler's intermediate language. The generator is implemented using QuickCheck (Claessen and Hughes 2000), and is able to leverage its extensive support for writing random test case generators. Writing a generator for well-typed terms in this context required significant effort, essentially implementing a function from types to terms in QuickCheck. The effort yielded significant benefit, however, as implementing the entire generator from the ground up provided many opportunities for specialized optimizations, such as variations of type rules that are more likely to succeed, or varying the frequency with which different constants are chosen. Pałka (2012) discusses the details.

Implementing this language in Redex was easy: the formal description in Pałka (2012) was ported directly into Redex with little difficulty. Once a type system is defined in Redex, the derivation generator can be immediately used to generate well-typed terms. Such an automatically derived generator is likely to make some performance tradeoffs versus a specialized one, and this comparison provided an excellent opportunity to investigate those.

The generators were compared by testing two of the properties used in Pałka (2012), and using same baseline version of the GHC (7.3.20111013) that was used there. **Property 1** checks whether turning on optimization influences the strictness of the compiled Haskell code. The property fails if the compiled function is less strict with optimization turned on. **Property 2** observes the order of evaluation, and fails if optimized code has a different order of evaluation compared to unoptimized code.

Counterexamples from the first property demonstrate erroneous behavior of the compiler, as the strictness of Haskell expressions should not be influenced by optimization. In contrast, changing the order of evaluation is allowed for a Haskell compiler to some extent, so counterexamples from the second property usually demonstrate interesting cases of the compiler behavior, rather than bugs.

Figure 27 summarizes the results of the comparison of the two generators. Each row represents a run of one of the generators, with a few varying parameters. Pałka (2012)'s generator as is referred to as "hand-written." It takes a size parameter, which was varied over 50, 70, and 90 for each property. "Redex poly" is the initial implementation of this system in the Redex, the direct translation of the language from Pałka (2012). The Redex generator takes a depth parameter, which we vary over 6, 7, 8, and, in one case, 10. The depths are chosen so that both generators target terms

Generator	**Terms/Ctrex.**	**Gen. Time (s)**	**Check Time (s)**	**Time/Ctrex. (s)**
Property 1				
Hand-written (size: 50)	25K	0.007	0.009	413.79
Hand-written (size: 70)	16K	0.009	0.01	293.06
Hand-written (size: 90)	12K	0.011	0.01	260.65
Redex poly (depth: 6)	∞	0.361	0.008	∞
Redex poly (depth: 7)	∞	0.522	0.009	∞
Redex poly (depth: 8)*	4000K	0.63	0.008	2549K
Redex non-poly (depth: 6)*	500K	0.038	0.008	23K
Redex non-poly (depth: 7)	668	0.082	0.01	61.33
Redex non-poly (depth: 8)	320	0.076	0.01	27.29
Property 2				
Hand-written (size: 50)	100K	0.005	0.007	1K
Hand-written (size: 70)	125K	0.007	0.008	2K
Hand-written (size: 90)	83K	0.009	0.009	2K
Redex poly (depth: 6)	∞	0.306	0.005	∞
Redex poly (depth: 7)	∞	0.447	0.005	∞
Redex poly (depth: 8)	∞	0.588	0.005	∞
Redex non-poly (depth: 6)	∞	0.059	0.005	∞
Redex non-poly (depth: 7)	∞	0.17	0.01	∞
Redex non-poly (depth: 8)	∞	0.142	0.008	∞
Redex non-poly (depth: 10)*	4000K	0.196	0.01	823K

Figure 27: Comparison of the derivation generator and a hand-written typed term generator. ∞ indicates runs where no counterexamples were found. Runs marked with * found only one counterexample, which gives low confidence to their figures.

of similar size.[1] (Figure 28 compares generated terms at targets of size 90 and depth 8). “Redex non-poly” is a modified version of the initial implementation, the details of which are discussed below. The columns show approximately how many tries it took to find a counterexample, the average time to generate a term, the average time to check a term, and finally the average time per counterexample over the entire run. Note that the goal type of terms used to test the two properties differs, which may affect generation time for otherwise identical generators.

[1]Although it is possible to generate terms of larger depth, the runtime increases quickly with the depth. One possible explanation is that well-typed terms become very sparse as term size increases. Grygiel and Lescanne (2013) show how scarce well-typed terms are even for simple types. Polymorphism seems to exacerbate this problem.

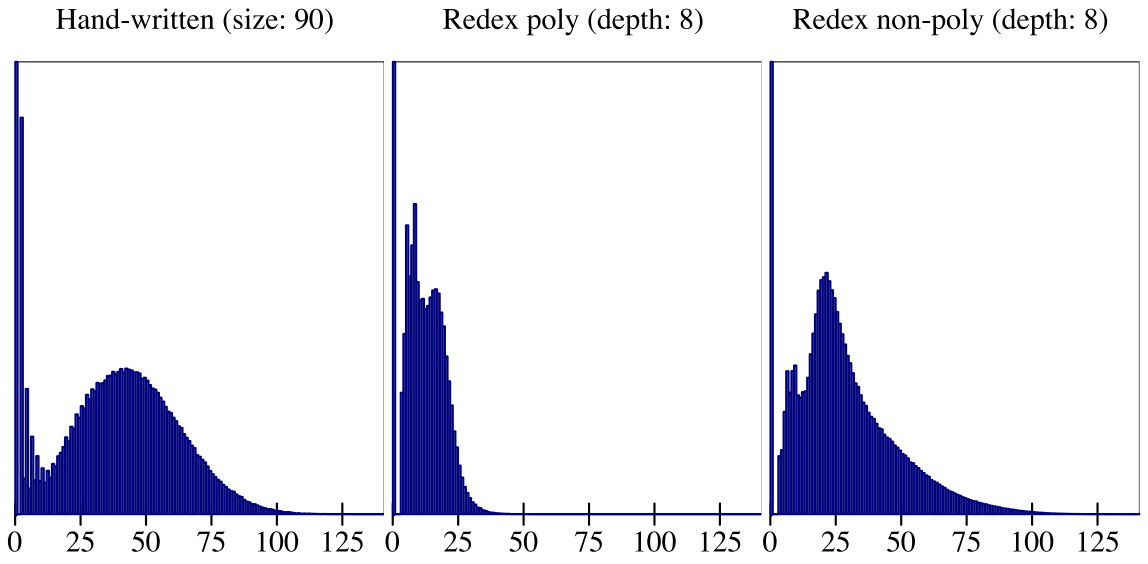


Figure 28: Histograms of the sizes (number of internal nodes) of terms produced by the different runs. The vertical scale of each plot is one twentieth of the total number of terms in that run.

A generator based on the initial Redex implementation was able to find counterexamples for only one of the properties, and did so and at significantly slower rate than the hand-written generator. The hand-written generator performed best when targeting a size of 90, the largest, on both properties. Likewise, Redex was only able to find counterexamples when targeting the largest depth on property one. There, the hand-written generator was able to find a counterexample every 12K terms, about once every 260 seconds. The Redex generator both found counterexamples much less frequently, at one in 4000K, and generated terms several orders of magnitude more slowly. Property two was more difficult for the hand-written generator, and the first try in Redex was unable to find any counterexamples there.

Comparing the test cases from both generators, we found that Redex was producing significantly smaller terms than the hand-written generator. The left two histograms in figure 28 compare the size distributions, which show that most of the terms made by the hand-written generator are larger than almost all of the terms that Redex produced (most of which are clumped below a size of 25). The majority of counterexamples produced with the hand-written generator fell in this larger range.

Digging deeper, it seemed that Redex's generator was backtracking an excessive amount. This directly affects the speed at which terms are generated, and it also causes the generator to fail more often because the search limits discussed in section 5.4 are exceeded. Finally, it skews the distribution toward smaller terms because these failures become more likely as the size of the search space expands. A reasonable hypothesis is that the backtracking was caused by making doomed choices when instantiating polymorphic types and only discovering that much later in the search, causing it to get stuck in expensive backtracking cycles. The hand-written generator avoids such problems by encoding model-specific knowledge in heuristics.

A variant Redex model was created to test this hypothesis, identical to the first except with a pre-instantiated set of constants, and removing all other polymorphism. The 40 most common instantiations of constants were selected from a set of counterexamples to both models generated by the hand-written generator. Runs based on this model are referred to as "Redex non-poly" in both figure 27 and figure 28.

As figure 28 shows, we get a much better size distribution with the non-polymorphic model, comparable to the hand-written generator's distribution. A look at the second column of figure 27 shows that this model produces terms much faster than the first try in Redex, though still slower than the hand-written generator. This model's counterexample rate is especially interesting. For property one, it ranges from one in 500K terms at depth 6 to, astonishingly, one in 320 at depth 8, providing more evidence that larger terms make better test cases. This success rate is also much better than that of the hand-written generator, and in fact, it was this model that was most effective on property 1, finding a counterexample approximately every 30 seconds, significantly faster than the hand-written generator. Thus, it is interesting that it did much worse on property 2, only finding a counterexample once every 4000K terms, and at very large time intervals. The reason for this discrepancy remains unknown.

Overall, the derivation generator is not competitive with the hand-tuned generator when it has to cope with polymorphism. Polymorphism is problematic because it requires the generator to make parallel choices that must match up, but where the generator does not discover that those choices must match until much later in the derivation. Because the choice point is far from the place where the constraint is discovered, the generator spends much of its time backtracking. The improvement in generation speed for the Redex generator when removing polymorphism provides evidence for the explanation of what makes generating these terms difficult. The ease with which this language could be implemented in Redex, and as a result, conduct this experiment, speaks to the value of a general-purpose generator, and of lightweight semantics tools.

CHAPTER 8

Related Work

Work related to the constraint solver is addressed in section 5.6, and studies on random testing most closely related to those of this work are discussed in section 6.1. This chapter discusses related work in random and property-based testing and its application in semantics engineering.

8.1. Property-based Testing

Quickcheck (Claessen and Hughes 2000) 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 (Pałka 2012). 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. Redex's approach, on the other hand, is to provide a straightforward way to implement a test generator. The relationship to Pałka's work is discussed in more detail in section 7.2, including a direct comparison on a few of the properties tested by Pałka (2012).

SmallCheck and Lazy SmallCheck (Runciman et al. 2008) are other Haskell libraries for property-based testing. They differ from QuickCheck in that they use exhaustive testing instead of random testing. Lazy SmallCheck is particularly successful, using partial evaluation to prune the space from which test cases are drawn based on the property under test. They also perform a comparative evaluation of SmallCheck, Lazy SmallCheck, and QuickCheck.

Perhaps the most closely related work is Claessen et al. (2014)'s typed term generator. Their work addresses specifically the problem of generating well-formed lambda terms based an implementation of a type-checker (in Haskell). They measured their approach against property 1 from section 7.2 and it performs better than Redex's 'poly' generator, but they are working from a lower-level specification of the type system. Also, their approach observes the order of evaluation of the predicate, and prunes the search space based on that; it does not use constraint solving.

Efficient random generation of abstract data types has seen some interesting advances in previous years, much of which focuses on enumerations. Feat (Duregard et al. 2012), 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 Pałka (2012).)

Another, more specialized, approach to enumerations was taken by Grygiel and Lescanne (2013). 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 (2012) 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 cannot be used for random generation because only bit-strings that have a prefix-closure property correspond to well-formed terms.

SciFe (Kuraj and Kuncak 2014) is a Scala library providing combinators that enable the construction of enumerations similar to those of section 3.2. Kuraj and Kuncak (2014) conduct a study comparing generation speed for 5 data structures with nontrivial invariants such a red-black trees or sorted lists. They compare their approach, the CLP approach described by Senni and Fioravanti (2012), and Korat (Boyapati et al. 2002), and find that their approach is the fastest at exhaustively generating structures up to a given size.

Senni and Fioravanti (2012) study the application of CLP to the exhaustive generation of several different data structures, including red-black trees and sorted lists. They report on a comparison with Korat (Boyapati et al. 2002), finding that their approach is faster than Korat at enumerating all inhabitants of such constrained types below a given size bound. They also include an in-depth discussion of how to efficiently implement CLP generators, including the application of several optimization passes.

Korat (Boyapati et al. 2002) is an approach to exhaustive testing in Java that uses a form of state-space filtering to generate data types satisfying general structural invariants in Java. The authors perform a study comparing its performance at generating all valid types of a certain size with the Allow Analyzer, an auotmated analysis tool for a relational specification language. The comparison is performed using red-black trees, binary heaps, and other data structures.

8.2. Testing and Checking Semantics

Random program generation for testing a semantics or programming language implementation is certainly not a new idea, and goes back as least to the "syntax machine" of Hanford (1970), a tool for producing random expressions from a grammar similar to the ad-hoc generation method of section 3.1. The tool was intended for compiler fuzzing, a common use for that type of random generation. Other applications of random testing to compilers throughout the years are discussed in the 1997 survey of Bourjarwah and Saleh (1997).

In the area of random testing for compilers, of special note is Csmith (Yang et al. 2011) a highly effective tool at 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. It had found over 300 bugs in mainstream C compilers as of 2011.

Cheney and Momigliano (2007) design an automated model-checking framework based on αProlog (Cheney and Urban 2004), a programming language based on nominal logic, designed for modeling formal systems. They advocate automating mechanized checking for semantics in a manner similar to this work, although their approach is different, performing exhaustive checking up to some bound on model size. They conduct a study demonstrating their approach's ability to find bugs in both the substitution function and the typing judgment of a small lambda calculus modeled in αProlog. For comparison, the bugs they evaluate in the substitution function are very similar to **stlc-sub** bugs 1 and 2 from the Redex benchmark, and the type judgment bugs are very similar to **stlc** or **poly-stlc** bugs 3 and 9, all of which were found by most generators in this paper in interactive time periods as well.

Other recent work also applies constraint logic programming to test programming language semantics and implementations. Dewey et al. (2014) conduct a study using CLP to generate Javascript programs with richer constraints than traditional grammar-based fuzzers, but less complex than full type soundness. They target specific variants of test cases, such as the use of prototype-based inheritance or combinations of `with` statements with closures. They perform a comparison with a baseline stochastic grammar generator, making a convincing case that CLP is an improvement for this type of language fuzzing. A related study (Dewey et al. 2015a) demonstrates that CLP can be competitive with the most efficient known methods for generating data structures such as red-black trees, skip lists, and B-trees. The same approach is used in Dewey et al. (2015b) to find bugs in the Rust typechecker, by specifying a system that will usually (but not always)[1] generate well-typed terms.

Isabelle/HOL (Nipkow et al. 2011) is a proof assistant equipped with a logic designed to support semantics modeling. Significant work has been done to equip Isabelle with automatic testing and checking capabilities similar to those in Redex, although in a proof-assistant as opposed to a lightweight modeling context. It has support for random testing via an implementation of QuickCheck (Berghofer and Nipkow 2004) and two methods of model checking, Nitpick (Blanchette and Nipkow 2010) and Refute (Weber 2008). Property-based testing in Isabelle

[1]For example, the specification of System F used as an example in the paper uses a definition of substitution that is not capture-avoiding, which simplifies implementation and generation speed at the cost of sometimes producing terms that are not well-typed.

has recently been extended to try a number of different strategies by Bulwahn (2012), adding exhaustive testing, symbolic testing, and a narrowing-based strategy. Bulwahn (2012) also conducts a study comparing the different methods of test-case generation, similar to that of this dissertation.

The K Framework (Rosu and Serbanuta 2010; Rosu and Serbanuta 2014) is a lightweight semantics modeling framework with sophisticated rewriting rules. It provides testing via executability (as in Redex) as a well as model checking in a linear temporal logic, symbolic execution, and verification based on reachability using matching logic. It has been used to model, test, and check/verify a number of different programming languages, including C (Ellsion 2012), Java (Bogdanas and Rosu 2015), and Javascript (Park et al. 2015).

CHAPTER 9

Conclusion

Mechanizing semantics gives programming languages researchers the ability to build models of real-world programming languages. A lightweight framework combined with property-based testing allows a semantics engineer to effectively and quickly develop their models, gaining confidence in their correctness and consistency with actual implementations before attempting a formal proof of correctness.

To support this approach to mechanization, this dissertation introduced a new approach to automatically derive generators for property-based testing from lightweight definitions: a derivation generator based on relation and function definitions. It also developed a benchmark suite to evaluate generator performance, and used that to learn about the relative strengths of four generators: ad-hoc recursive generators, two enumeration based generators derived from grammars, and the derivation generator. The evaluation shows that all of them effectively find counterexamples for realistic properties of real-world semantics models in reasonable time frames, and that generating well-typed terms is significantly more effective. Overall, the evidence shows that automated checking based on lightweight definitions is a productive avenue toward improved tools for semantics engineering.

9.1. Future Work

The derivation generation approach proves to be the most effective when it can be applied, but it cannot be used on all Redex models. As already noted, there are a number of different features of Redex that are commonly used yet are not supported by the derivation generation approach. Addressing these issues, either by extending the derivation generation approach or by developing a new generator better, would be a productive direction for further research.

Redex's repeat patterns, or ellipses, a pattern language element analagous to the Kleene star, are problematic for the derivation generator because of the challenges involved in creating a constraint solver capable of handling them. Kutsia (2002)'s work on sequence unification, which handles patterns similar to Redex's, shows how a similar constraint solving algorithm looks, and should provide a good starting point for extending the constraint solver of section 5.3 to support repeat patterns. That would enable the derivation generator to support all definitions except those involving unquote.

Unquotes, or escapes from Redex to Racket, are even more problematic for an approach such as the derivation generator, as they allows arbitrary computation. There are at least two approaches to attempt to use definitions that include unquote as a basis for generators. One way to would be to move operations that are commonly used in unquote, such as integer arithmetic, list operations, or symbolic manipulations, into Redex, and extend the constraint solver to handle them as well. Another would be to design a new generator. A possibility that seems promising would be something along the lines of Claessen et al. (2014), pruning a search space based on the behavior of a predicate, which could be an arbitrary Racket function.

Bibliography

Claire E. Alvis, Jeremiah J. Willcock, Kyle M. Carter, William E. Byrd, and Daniel P. Friedman. cKanren: miniKanren with Constraints. In *Proc. Scheme and Functional Programming*, 2011.

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.

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

Stefan Berghofer and Tobias Nipkow. Random testing in isabelle/hol. In *Proc. IEEE Intl. Conf. Software Engineering and Formal Methods*, 2004.

Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relation Model Finder. In *Proc. Intl. Conf. Interactive Theorem Proving*, Lecture Notes in Computer Science volume 6172, pp. 131–146, 2010.

Denix Bogdanas and Grigore Rosu. K-Java: A Complete Semantics of Java. In *Proc. ACM Symp. Principles of Programming Languages*, 2015.

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

Chandrasekhar Boyapati, Sarfraz Khursid, and Darko Marinov. Korat: Automated Testing based on Java Predicates. In *Proc. Intl. Symp. Soft. Testing and Analysis*, 2002.

Lukas Bulwahn. The New Quickcheck for Isabelle: Random, Exhaustive and Symbolic Testing under One Roof. In *Proc. Lecture Notes in Computer Science*, 2012.

Lukas Bulwahn. Counterexample Generation for Higher-Order Logic Using Functional and Logic Programming. PhD dissertation, Technische Univeristat Munchen, 2013.

William E. Byrd. Relational Programming in miniKanren: Techniques, Applications, and Implementations. PhD dissertation, Indiana University, 2009.

James Cheney and Alberto Momigliano. Mechanized Metatheory Model-Checking. In *Proc. Intl. Conf. Principles and Practice of Declarative Programming*, pp. 75–86, 2007.

James Cheney and Christian Urban. αProlog: A Logic Programming Language with Names, Binding, and α-Equivalence. In *Proc. Intl. Conf. Logic Programming*, Lecture Notes in Computer Science volume 3132, pp. 269–283, 2004.

Koen Claessen, Jonas Duregard, and Michal H. Palka. Generating Constrained Random Data with Uniform Distribution. In *Proc. Intl. Symp. Functional and Logic Programming*, pp. 18–34, 2014.

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.

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

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

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

Kyle Dewey, Lawton Nichols, and Ben Hardekopf. Automated Data Structure Generation: Refuting Common Wisdom. In *Proc. Intl. Conf. Soft. Eng.* , 2015a.

Kyle Dewey, Jared Roesch, and Ben Hardekopf. Fuzzing the Rust Typechecker Using CLP. In *Proc. Intl. Conf. Automated Software Engineering*, 2015b.

Kyle Dewey, Jared Roesch, and Ben Hardekpf. Language Fuzzing Using Constraint Logic Programming. In *Proc. Intl. Conf. Automated Software Engineering*, 2014.

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

Chucky Ellsion. A Formal Semantics of C with Applications. PhD dissertation, University of Illinois, 2012.

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

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

Arjun Guha, Claudia Saftiou, and Shriram Krishnamurthi. The Essence of JavaScript. In *Proc. Euro. Conf. Object-Oriented Programming*, 2010.

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

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.

Andrew J. Kennedy and Dimitrios Vytiniotis. Every bit counts: The binary representation of typed data and programs. *J. Functional Programming* 22, pp. 529–573, 2012.

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

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

Casey Klein and Robert Bruce Findler. Randomized Testing in PLT Redex. In *Proc. Scheme and Functional Programming*, pp. 26–36, 2009.

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

Casey Klein, Matthew Flatt, and Robert Bruce Findler. The Racket Virtual Machine and Randomized Testing. 2011. `http://plt.eecs.northwestern.edu/racket-machine/`

Ivan Kuraj and Victor Kuncak. Scife: Scala Framework for Efficient Enumeration of Data Structures with Invariants. In *Proc. Scala Workshop*, pp. 45–49, 2014.

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.

Lee Naish. Adding Equations to NU-Prolog. In *Proc. Intl. Symp. Programming Language Implementation and Logic Programming*, 1991.

Max S. New. Enumerations. 2014. `docs.racket-lang.org/data/Enumerations.html`

Max S. New, Burke Fetscher, Jay McCarthy, and Robert Bruce Findler. Fair Enumeration Combinators. To Appear, 2015.

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. Isabelle/HOL—A Proof Assistant for Higher-Order Logic. Springer Verlag, 2011.

Daejun Park, Andrei Stefanescu, and Grigore Rosu. KJS: A Complete Formal Semantics of JavaScript. In *Proc. ACM Conf. Programming Language Design and Implementation*, 2015.

Michał H. Pałka. Testing an Optimising Compiler by Generating Random Lambda Terms. Licentiate dissertation, Chalmers University of Technology, Göteborg, 2012.

Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proc. International Workshop on Automation of Software Test*, 2011.

Joe Gibbs Politz, Matthew J. Carroll, Benjamin S. Lerner, Justin Pombrio, and Shriram Krishnamurthi. A Tested Semantics for Getters, Setters, and Eval in Javscript. In *Proc. Dynamic Languages Symposium*, 2012.

Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipthu, and Shriram Krishnamuthi. Python: The Full Monty, A Tested Semantics for the Python Programming Language. In *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages, and Applications*, 2013.

Girgore Rosu and Traian Florin Serbanuta. An Overview of the K Semantic Framework. *Journal of Logiac and Algebraic Programming* 79(6), pp. 397–434, 2010.

Grigore Rosu and Traian Florin Serbanuta. K Overview and SIMPLE Case Study. In *Proc. Intl. K Workshop*, ENTCS volume 304, pp. 3–56, 2014.

Celine Rouveirol. Flattening and Saturation: Two Representation Changes for Generalization. *Machine Learning*, 1994.

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. SmallCheck and Lazy SmallCheck: Automatic Exhaustive Testing for Small Values. In *Proc. ACM SIGPLAN Haskell Wksp.*, 2008.

Valerio Senni and Fabio Fioravanti. Generation of test data structures using constraint logic programming. In *Proc. Intl. Conf. Tests and Proofs*, 2012.

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

Tjark Weber. SAT-based Finite Model Generation for Higher-Order Logic. PhD dissertation, Technische Universität München, 2008.

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

Correctness of the Constraint Solver

A conjunction of equations ($\wedge$ *e* ...) is satisfiable if there is a substitution that makes all of the equations identical.

A disequational constraint $\delta = (\forall\ (x\ ...)\ (\vee\ (p_l \neq p_r)\ ...))$ is satisfiable if there is a substitution α, with $Vars(\alpha) \cap \{x\ ...\} = \emptyset$, such that for some p_l, p_r, there does not exist a substitution β where $\beta\alpha p_l$ is identical to $\beta\alpha p_r$. (In this section substitutions are applied by prepending them to the term.)

For some $C = (\wedge\ (\wedge\ e\ ...)\ (\wedge\ \delta\ ...))$, C is consistent if there is a substitution α that makes both sides of all equations *e* identical, and for all disequational constraints δ, the result $\alpha\delta$ of applying α to δ remains satisfiable.

A conjunction of equations ($\wedge$ *e* ...) is in canonical form if $(\wedge\ e\ ...) = (\wedge\ (x = p)\ ...)$, where if $x_l \in \{x\ ...\}$, then $x_l \cap Vars((p\ ...)) = \emptyset$. Note that the equations themselves express an idempotent substitution that makes the equations identically true, i.e. $\{(x := p)\ ...\}$, so the equations are immediately satisfiable.

Finally, $C = (\wedge\ (\wedge\ (x = p)\ ...)\ (\wedge\ \delta\ ...))$ is in canonical form if the equations are in canonical form, and if α is the substitution expressed by the equations (as above), $\alpha(\delta\ ...) = (\delta\ ...)$, and for each $\delta = (\forall\ (x_\delta\ ...)\ (\vee\ (p_l \neq p_r)\ ...))$, there exists $p_l = x_p$, and $x_p \cap \{x_\delta\ ...\} = \emptyset$, and $p_r \cap \{x_\delta\ ...\} = \emptyset$, i.e. at least one of the inequations in the disjunction has a left hand side that is a variable which is not in the domain of the substitution expressed by the equations and is not universally quantified, and a right hand side that is not a universally quantified variable. As above, we have $\alpha x_p = x_p$, and we can choose β such that there is no substitution γ where $\gamma\beta\alpha x_p$ and $\gamma\beta\alpha p_r$ are identical. (If p_r

is a variable, it is unconstrained, otherwise it is a constructor, and in either case we can choose to make $\beta\alpha x_p$ and $\beta\alpha p_r$ conflict.) This gives us:

Lemma 1. *If C is in canonical form, C is consistent.*

This justifies the use of check in solve, which simply verifies that the disequational part of *C* is in canonical form. That the equational portion of *C* is in canonical form is a property of unify.

We also need a few definitions regarding substitutions. Two substitutions α and β are equal $\alpha = \beta$ if for any any variable x, $\alpha x = \beta x$. A substitution α is more general than β, written $\alpha \leqslant \beta$, if there exists some substitution γ such that $\beta = \gamma\alpha$. A substitution α unifies two terms s and t if $\alpha s = \alpha t$ (where $=$ means they are syntactically identical). Finally, if α is a unifier of s and t, and for every unifier β of s and t, $\alpha \leqslant \beta$, then α is a most general unifier (mgu) of s and t. The notions of unifier and mgu are extended naturally to sets of equations.

A standard result regarding syntactic unification adapted to this setting (see, for example, Baader and Snyder (2001)) is:

Theorem 1. *For any equations* (e_0 ...) *and* ($\wedge$ e ...) *in canonical form,* unify$[\![$((e_0 ...)), ($\wedge$ e ...)$]\!]$ *terminates with* $\perp$ *if there is no unifier of* (e_0 ...) *and the equations* ($\wedge$ e ...)*. Otherwise, it terminates with* ($\wedge$ e_Ω ...) *in canonical form, such that the substitution expressed by* ($\wedge$ e_Ω ...) *is an mgu of e and* ($\wedge$ e ...)

We now prove some lemmas that justify the use of unify to simplify disequational constraints δ in disunify.

Lemma 2. *If a substitution* α *is idempotent then* $\beta = \beta\alpha \Leftrightarrow \alpha \leqslant \beta$.

Proof. The forward direction holds by definition. For the reverse direction, by definition there must be some γ such that $\gamma\alpha = \beta$, so, for any x,

$$\beta\alpha x = \gamma\alpha\alpha x = \gamma\alpha x = \beta x$$

where the middle equality depends on the idempotency of α. □

Lemma 3. *If* unify⟦$(e\ ...)$, $(\wedge)$⟧ $= ((x = p)\ ...)$*, then for any unifier* θ *of* $(e\ ...)$*, for any* x_i *and* p_i *paired in* $((x = p)\ ...)$*,* $\theta x_i = \theta p_i$*.*

Proof. If γ is the mgu expressed by $((x = p)\ ...)$, then since $\gamma \leqslant \theta$ and γ is idempotent, $\theta = \theta\gamma$ (Lemma 1), so $\theta x_i = \theta\gamma x_i = \theta p_i$. □

Lemma 4. *If* unify⟦$(e\ ...)$, $(\wedge)$⟧ $= ((x = p)\ ...)$*, then for any substitution* ω*,*

$$\text{unify}⟦(\omega e\ ...), (\wedge)⟧ = \bot \Leftrightarrow \text{unify}⟦((\omega x = \omega p)\ ...), (\wedge)⟧ = \bot.$$

Proof. For the forward direction, we know there cannot be a unifier for the equations $(\omega e\ ...)]$. Now suppose there were some unifier ρ for $(\omega(x = p)\ ...)$. Then $\rho(\omega x\ ..) = \rho(\omega p\ ..)$, and if γ is the mgu for $(e\ ...)$, then $\rho\omega\gamma(x\ ...) = \rho\omega(p\ ...) = \rho\omega(x\ ...)$. That implies that $\gamma \leqslant \rho\omega$, so $\rho\omega$ unifies $(e\ ...)$, a contradiction. So there can be no such ρ.

For the reverse direction, by Lemma 2 any unifier ρ of $(e\ ...)$ must make all the equations $\rho((x = p)\ ...)$ identical, so if there is no unifier of $((x = p)\ ...)$, there can be none for $(e\ ...)$. □

That means that if some set of disequations $(\vee\ (p_1 \neq p_2)\ ...)$ is satisfiable using substitution ω, then if unify⟦$((p_1 = p_2)\ ...)$, $(\wedge)$⟧ $= ((x = p)\ ...)$, the disequations $(\vee\ (x \neq p)\ ...)$ are satisfiable by the exact same substituion ω. That justifies the use unify as a simplification in disunify.

After simplifying the disequations, param-elim is applied to restrict the satisfying substitution with respect to the quantifier. In the following, for a disequational constraint of the form

($\forall$ (x ...) (($p_1 \neq p_2$) ...)), we will refer to the set of parameters $\{x...\}$ as X. We refer to a substitution α that makes the equations (e ...) $=$ (($p_1 = p_2$) ...) impossible to satisfy (and thus satisfies the disequation) as an *excluding* substitution. The excluding substitution must also satisfy the constraint that $Variables(\alpha) \cap X = \varnothing$. The following lemmas justify the steps used by param-elim.

Lemma 5. *If* (e ...) $=$ (e_1 ... ($x = p$) e_2 ...)*, and* $x \in X$*, then* α *excludes* (e ...) *iff* α *excludes* (e_1 ... e_2 ...).

PROOF. We assume (e ...) has the property that x does not occur in (e_1 ... e_2 ...), since it is the result of unification and corresponds to an idempotent substitution. Since $\alpha x = x$, we know that unify$[\![(x = \alpha p), (\wedge)]\!] \neq \bot$. Thus if α excludes (e ...), it must exclude (e_1 ... e_2 ...). Clearly if α excludes (e_1 ... e_2 ...), it excludes (e ...). □

In the following lemma, for convenience we refer to a list of equations in canonical form (($x = p$) ...) as a set $\{x_1 = p_1\}\{x_2 = p_x\}...\{x_n = p_n\}$, where juxtaposition means union.

Lemma 6. *If* $\gamma = \{x_1 = x\}...\{x_n = x\}\gamma'$*, where* $y_i \notin X$ *and* $x \in X$*, then* α *excludes* γ *iff* α *excludes* $\{y_i = y_j | 1 \leqslant i, j \leqslant n\} \cup \gamma'$.

PROOF. It must be the case that α excludes at least one equation $x_i = x_j$ for some i, j. But if unify$[\![((\alpha x_i = \alpha x_j)), (\wedge)]\!] = \bot$, then α excludes $\{x_1 = x\}...\{x_n = x\}$, because there is no value for αx that can satisfy the equations. Thus if α excludes $\{y_i = y_j | 1 \leqslant i, j \leqslant n\} \cup \gamma'$, it excludes γ. If unify$[\![((\alpha x_i = \alpha x)\ (\alpha x_j = \alpha x)), (\wedge)]\!] = \bot$, and $\alpha x = x$, since the right-hand sides will be identical, it must be the case that unify$[\![((\alpha x_i = \alpha x_j)), (\wedge)]\!] = \bot$, and α excludes $x_i = x_j$. □

The above lemma refers to the step in param-elim that makes use of elim-x. The above two lemmas show that param-elim preserves the satisfiability criteria for a disequational constraint δ.

Lemma 7. param-elim *terminates.*

Proof. Every recursive call decreases the number of equations that have a left or right hand side that is a single parameter. (Parameters are the list of variables in param-elim's second argument.) □

The termination of all other functions in the constraint solver is obvious.

Lemma 8. *Given a disequation δ,* disunify$[\![\delta]\!]$ *terminates with $\perp$ if δ is unsatisfiable or $\top$ if δ is always satisfiable; otherwise it terminates with a disequation in canonical form that is equivalent (satisfiable by the same substitutions) to δ.*

Proof. Follows directly Lemmas 4, 5, 6, and 7. □

The correctness of the constraint solver follows directly from the correctness of unify, disunify, and check:

Theorem 2. *For any e and C in canonical form,* solve *terminates with $\perp$ if there is no unifier of e and the equations in C that preserves the consistency of C. Otherwise, it terminates with C_Ω in canonical form, such that the equations in C_Ω are an mgu of e and the equations in C, and C_Ω is consistent.*

Theorem 3. *For any δ and C in canonical form,* dissolve *terminates with $\perp$ if δ and C are inconsistent. Otherwise, it terminates with C_Ω in canonical form, such that the disequations in C_Ω are satisfiable iff the union of those in C and δ is, and C_Ω is consistent.*

APPENDIX B

Detailed Listing of Benchmark Bugs

Model	Bug#	S/M/D/U	Size	Description of Bug
stlc	1	S	3	app rule the range of the function is matched to the argument
	2	M	5	the ((cons v) v) value has been omitted
	3	S	8	the order of the types in the function position of application has been swapped
	4	S	9	the type of cons is incorrect
	5	S	7	the tail reduction returns the wrong value
	6	M	7	hd reduction acts on partially applied cons
	7	M	9	evaluation isn't allowed on the rhs of applications
	8	U	12	lookup always returns int
	9	S	15	variables aren't required to match in lookup
poly-stlc	1	S	6	app rule the range of the function is matched to the argument
	2	M	11	the (([cons @ τ] v) v) value has been omitted
	3	S	14	the order of the types in the function position of application has been swapped
	4	S	15	the type of cons is incorrect
	5	S	16	the tail reduction returns the wrong value
	6	M	16	hd reduction acts on partially applied cons
	7	M	9	evaluation isn't allowed on the rhs of applications
	8	U	15	lookup always returns int
	9	S	18	variables aren't required to match in lookup
stlc-sub	1	S	8	forgot the variable case
	2	S	13	wrong order of arguments to replace call
	3	S	10	swaps function and argument position in application
	4	D	22	variable not fresh enough
	5	SM	17	replace all variables
	6	S	8	forgot the variable case
	7	S	13	wrong order of arguments to replace call
	8	S	10	swaps function and argument position in application
	9	SM	17	replace all variables
let-poly	1	S	8	use a lambda-bound variable where a type variable should have been

	2	D	28	the classic polymorphic let + references bug
	3	M	3	mix up types in the function case
	4	S	8	misspelled the name of a metafunction in a side-condition, causing the occurs check to not happen
	5	M	3	eliminate-G was written as if it always gets a Gx as input
	6	M	6	$\vee$ has an incorrect duplicated variable, leading to an uncovered case
	7	D	12	used let --> left-left-λ rewrite rule for let, but the right-hand side is less polymorphic
list-machine	1	S	22	confuses the lhs value for the rhs value in cons type rule
	2	M	22	var-set may skip a var with matching id (in reduction)
	3	S	29	cons doesn't actually update the store
rbtrees	1	M	13	ins does no rebalancing
	2	M	15	the first case is removed from balance
	3	S	51	doesn't increment black depth in non-empty case
delim-cont	1	M	46	guarded mark reduction doesn't wrap results with a list/c
	2	M	25	list/c contracts aren't applied properly in the cons case
	3	S	52	the function argument to call/comp has the wrong type
rvm	2	M	24	stack offset / pointer confusion
	3	D	33	application slots not initialized properly
	4	M	17	mishandling branches when then branch needs more stack than else branch; bug in the boxenv case not checking a stack bound
	5	M	23	mishandling branches when then branch needs more stack than else branch; bug in the let-rec case not checking a stack bound
	6	M	15	forgot to implement the case-lam branch in verifier
	14	M	27	certain updates to initialized slots could break optimizer assumptions
	15	S	21	neglected to restrict case-lam to accept only 'val' arguments