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

1 Overview of PLT Redex and Reduction Semantics

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

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

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

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

```
#f
```

The `define-language` form is used to describe a grammar and is usually the first element in a Redex program, since most other operations in Redex take place with respect to some language. It defines the language and binds it to an identifier, in this case

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

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

Grammar

$$\begin{aligned}
e &::= \ldots. \\
&\quad \mid (\text{if0}\ e\ e\ e) \\
&\quad \mid (+\ e\ e) \\
v &::= (\lambda\ (x\ \tau)\ e) \mid n \\
E &::= (E\ e) \mid (v\ E) \mid (+\ E\ e) \mid (+\ v\ E) \\
&\quad \mid (\text{if0}\ E\ e\ e) \mid []
\end{aligned}$$

Reduction relation

$$\begin{aligned}
E[((\lambda\ (x\ \tau)\ e)\ v)] &\longrightarrow E[e\{x \leftarrow v\}] && [\beta] \\
E[(\text{if0}\ 0\ e_1\ e_2)] &\longrightarrow E[e_1] && [\text{if-0}] \\
E[(\text{if0}\ n\ e_1\ e_2)] &\longrightarrow E[e_2] && [\text{if-n}] \\
&\qquad \text{where } n \neq 0 \\
E[(+\ n_1\ n_2)] &\longrightarrow E[\ulcorner n_1 + n_2 \urcorner] && [\text{plus}]
\end{aligned}$$

Metafunctions

$$\begin{aligned}
\text{lookup}[\![(x\ \tau\ \Gamma), x]\!] &= \tau \\
\text{lookup}[\![(x_1\ \tau\ \Gamma), x_2]\!] &= \text{lookup}[\![\Gamma, x_2]\!] \\
\text{lookup}[\![\bullet, x]\!] &= \#\text{f}
\end{aligned}$$

Type judgment

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

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

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

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

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

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

Evaluation

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

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

our language is bound to STLC. After the language identifier comes a list of nonterminal definitions, each of which means that the nonterminal (left of ::=) may be satisfied by some set of patterns (right of ::=). Patterns are lists built from Racket constants, built-in Redex-patterns, or non-terminals. Here number and variable-not-otherwise-mentioned are built-in Redex patterns that respectively match Racket numbers and symbols that do not appear in productions in the grammar (λ is excluded, for example). Once a language is defined, the redex-match form can be used to ask Redex to attempt to match a pattern of literals and non-terminals against some term:

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

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

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

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

```
> (redex-match STLC (in-hole E (+ n_1 n_2))
               (term ((λ (x num) x) (+ 1 2))))
(list
 (match
  (list
   (bind 'E '((λ (x num) x) hole))
   (bind 'n_1 1)
   (bind 'n_2 2))))
```

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

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

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

The reduction relation, pictured below the grammar in figure 1, describes how computation takes place. The relation is described as a set of reduction rules, each of which matches a specific scheme of expressions on the left hand side of the arrow, and performs some transformation to a new scheme on the right hand side. Each rule is meant to describe a single step of computation. The left hand side of a rule is a *pattern*, which attempts to parse a *term* according to the grammar, and if it succeeds, the non-terminals of the pattern are bound for use in the right hand to reconstruct a new term. Thus the relation pairs terms which match the left hand side to terms constructed by the right hand side, given the bound non-terminals from the match.

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

The reduction relation can be formalized in Redex as:

```
(define STLC-red
  (reduction-relation STLC
   (--> (in-hole E ((λ (x τ) e) v))
        (in-hole E (subst e x v))
        β)
   (--> (in-hole E (if0 0 e_1 e_2))
        (in-hole E e_1)
        if-0)
   (--> (in-hole E (if0 n e_1 e_2))
        (in-hole E e_2)
        (side-condition (term (different n 0)))
        if-n)
   (--> (in-hole E (+ n_1 n_2))
        (in-hole E (sum n_1 n_2))
        plus)))
```

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

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

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

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

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

It is useful to describe computation in terms of individual steps for accuracy, but ultimately a semantics is interested in the final result of evaluating and expression, or an answer. This is expressed by taking the transitive-reflexive closure of the reduction relation, denoted by $\rightarrow^*$. If the transitive-reflexive closure of the reduction relates an expression e_1 to an expression e_2 that is not in the domain of the reduction relation (does not take a step) then we say that the e_1 reduces to e_2, or that e_2 is a normal form[4] of e_1. Redex allows users to reduce expressions completely through the `apply-reduction-relation*` form, which returns a list of the final terms in all non-terminating reduction paths originating from its argument (when this is possible).[5]

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

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

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

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

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

Of course, *Eval* is unfortunately not defined for all expressions that satisfy the e non-terminal of our grammar. The evaluation of some expressions may terminate in

[4]Not all expression have normal forms, and it is usually desirable that normal forms of allowable expressions other properties (i.e., are values). We use a type system to address these issues, which is described shortly.

[5]Although not detailed here, Redex also has support for exploring the details of reduction graphically.

non-values (known as "stuck" states), and some evaluations may never terminate. Non-terminating expressions are useful, but expressions that terminate in non-values don't have a defined answer and should be considered errors. We can apply a type system to eliminate expressions that result in stuck states.[6] Type systems are relations between expressions and *types*, which denote what kind of value an expression will evaluate to. Thus an expression that has a type will reduce to a value. Types in our language match the τ non-terminal, and can correspond to either numbers ("num" types) or functions ("arrow" types). The type system is delineated by the inference rules of figure 1, which inductively describe a ternary relation between type environments Γ, expressions, and types. The statements above the horizontal lines are premises of each inference, and that on the bottom is the conclusion. This type of relation is known as a "judgment form", since it is typically used to make a decision about some syntactic object. (See ??? (???) for more on judgment forms and type systems.) Judgment forms are typically used in practice by starting with a conclusion we wish to prove and attempting to construct a derivation of that conclusion. In this case one might start with an expression and try to construct a derivation that relates it to some type in the empty environment.[7] A "well-typed" expression is precisely that, one that satisfies the type judgment for some type in an empty environment, i.e. the expression *e* is well-typed if $\bullet \vdash e : \tau$, for some τ.

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

```
(define-judgment-form STLC
  #:mode (tc I I O)
  [--------------
   (tc Γ n num)]
  [(where τ (lookup Γ x))
   ----------------------
   (tc Γ x τ)]
  [(tc (x τ_x Γ) e τ_e)
   ---------------------------------
   (tc Γ (λ (x τ_x) e) (τ_x → τ_e))]
  [(tc Γ e_1 (τ_2 → τ)) (tc Γ e_2 τ_2)
   -----------------------------------
   (tc Γ (e_1 e_2) τ)]
  [(tc Γ e_0 num)
   (tc Γ e_1 τ) (tc Γ e_2 τ)
   ---------------------------
   (tc Γ (if0 e_0 e_1 e_2) τ)]
  [(tc Γ e_0 num) (tc Γ e_1 num)
   ------------------------------
```

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

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

```
     (tc Γ (+ e_0 e_1) num)])
```

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

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

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

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

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

The lookup metafunction takes a variable x and an enviroment Γ and first tries to match x to the beginning of the environment, returning the corresponding type τ if it succeeds. Otherwise, it recurs on what remains of Γ. The final case indicates that lookup fails and returns *#f*, or *false*, when passed an empty environment. Metafunctions are functions on and in the object language but are defined in the meta-language (the

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

language of the semantics: in this case, Redex) – in Redex, metafunctions may only be used inside of `term` (including implicit uses of `term`, such as the right-hand sides of reduction relations or metafunctions themselves). For example:

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

2 Random Testing in Redex

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

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

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

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

First, let's examine how we might test our system in actuality using Redex's testing infrastructure. We may wish, for example, to test that expressions in our language do not result in stuck states. To do so, one might write a predicate checking for that property, and then generate a collection of terms and check that the predicate is never false. This is a common situation, so Redex provides `redex-check`, a form that conducts such testing automatically, given a user-provided predicate. The following code checks a property asserting that terms always reduce to values:

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

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

```
> (redex-check STLC e
               (or (not (judgment-holds (tc • e τ)))
                   (redex-match STLC v
                                (car (apply-reduction-relation*
                                      STLC-red
                                      (term e))))))
redex-check: no counterexamples in 1000 attempts
```

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

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

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

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

The result reports that the "`plus`" reduction rule was used 3 times over the 1000 test cases, and none of the other rules was used even once. Clearly, the vast majority of the terms being generated are poor tests. To give an idea of what kind of terms are generated, figure 2 shows some statistics for random terms in this language, and exposes some of the difficulty inherent in generating "good" terms. Although about 10% of random expresssions are well typed, only 0.5% are well-typed and not a constant or a constant function (a function of the form *(λ (x τ) n)*). The terms that are good tests for the property in question, those that are well-typed and exercise the reduction, are even rarer, at 0.26% of all terms. Thus it is unsurprising that test coverage is so poor.

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

The strategy of using strategically selected source patterns and preprocessing terms in some way is typical of most serious testing efforts involving Redex, and has been effective in many cases. It has been used to successfully find bugs in a Scheme semantics (??? ???), the Racket Virtual Machine (??? ???), and various language models drawn from the International Conference on Functional Programming (??? ???). However, given the apparent rarity of useful terms even in a language as simple as our small lambda calculus example, one may wonder if there is not a more effective term generation strategy that requires less work and ingenuity from the user.

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