Semantics Engineering with Redex in Racket

Adrian Manea
Coordinator: Assoc. Prof. Traian Şerbănuţă

March 9, 2020

Contents

TO ADD/CLARIFY

INTRODUCTION AND MOTIVATION

The starting point for the present work was my motivation of learning the programming language Racket. Having a background in mathematics, the lambda calculus and the function programming languages appealed to me instantly and once I got acquainted with Emacs and the Lisps, my interest grew. Furthermore, the excellent theoretical foundations laid by J. McCarthy in [McC62] and [McC61], described by some computer scientists as having introduced a paradigm shift in theoretical computer science that's comparable to the non-Euclidean geometry revolution proved to be an excellent introduction and motivation for wanting to learn more about the Lisp family of programming language. Before long, I discovered the influential [AS96] and saw how an apparently simple programming language such as Scheme can reveal wonderful constructions of various degrees of abstraction.

Doing some more reading in the world of Lisp dialects, I have discovered Racket, whose appeal was instantaneous to me, since it is so often described not as a general purpose programming language derived from Scheme, but rather as a toolbox for constructing languages to solve various problems. In fact, as I saw it, it offers the necessary items for one to properly understand, craft and teach languages that exhibit particular behaviour.

This is precisely the aim of the current work. Given the inherent shortcomings of Scheme, assumed by its creators for the sake of simplicity and ease of use and extension, and not trying to delve into the whole "zoo" of Lisp dialects, I found Racket to be the most appropriate object of study for my purposes. These are to study semantic aspects of (mostly) functional programming languages, as well as type theory and related formal methods, which Racket has the flexibility to allow, all while using a mild variation of the syntax that was established ever since McCarthy's (Common) Lisp.

However, this work is not a Racket manual and given its sheer flexibility and array of features, it is beyond my scope to thoroughly explore this language-toolbox, at least not when confined within the scope of this dissertation. Therefore, the precise topic I chose to focus on this work is that of **PLT Redex** (henceforth called **Redex** in short, since PLT is the name of the research group that started it, continuing efforts from PLT Scheme, which was to become Racket).

Inspired by the great article and presentation of B. Findler at POPL 2012 ([FF+12]), I will be presenting the main features of Redex (in Racket, as it is implemented), provide some examples and try to show how it can be used not only to create toy languages, but also some which come with included formal methods of checking correctness (a rather vague term which will be detailed in due time).

The plan of the work is as follows. A short historical presentation and preliminaries make up the first part of the dissertation. Then a Scheme and Racket *crash course* will follow, focusing on specific aspects that will come in handy when discussing Redex, such as `call/cc` and `amb`. The main part of the work will then contain the actual presentation of Redex, along with some standard examples that the authors provided ([amb20, lon20, FF09]). Finally, further examples are provided, which exhibit features of interest, related mostly to type theory.

The work is in fact part of a more elaborate plan, which will be detailed in the final section of this dissertation.

It is also worth mentioning that the PLT group ([plt20]) used Redex as a starting point for their work on a language-creating toolbox and their current focus is on Pyret, described as *a programming language designed to serve as an outstanding choice for programming education while exploring the confluence of scripting and functional programming* ([Gro20]). However, since as we will detail, the wider focus is on Racket, we will not touch on this subject here.

change the bib style (especially to go well with websites)

CHAPTER 1

HISTORY AND PRELIMINARIES

index!

marginpars?

Ever since the introduction of the lambda calculus formalism for representing functions, by A. Church and others in the 1930s, various uses of it in the foundations of mathematics and in theoretical computer science arose. As it is mentioned in [McC61] and [McC62], while unsuited for recursive functions representation, lambda calculus can be extended to help represent a generous amount of key concepts of programming.

Once the technology started to develop to support the theoretical advances in computer science, approaches such as Landin's in [Lan64] and McCarthy's in [McC60] showed how one can "mechanically" implement some variants of lambda calculus in order to create what were to become prototypes of functional programming languages. This happened in an era when the procedural and imperative ALGOL was one of the greatest achievements in programming languages.[1]

Essentially, some of the features that McCarthy emphasized and which became included later in Common Lisp and Scheme were *continuations*, as a means of capturing the computational content of a lambda-expression, thus making it more useful for recursive functions and *ambiguous "functions"*. The latter were thought of as a means of representing non-determinism and are not actual functions, since they can return either nothing or one or many results. Without getting more into the historical details, we will detail these features in the Scheme crash course a bit later.

reference the section

Making a 50-year time leap, at the POPL conference in 2012, the PLT group, voiced by B. Findler and M. Felleisen presented a tool (rather, a tool*box*) called **Redex** for programming language

[1] McCarthy explains in some detail how his Lisp language (so called by shrinking "list processing") is intended to differ and be better than ALGOL, COBOL and UNCOL in his introduction of [McC61], pp. 1–4.

creation which proposes an approach that is similar to any software development. They called the approach *semantics engineering* and expanded on the idea in their book [FF09]. Basically, what the presentation of [FF+12] outlined and was detailed in the book cited previously were two key aspects which they proposed:

- firstly, they emphasized the empowering of programmers with *language creation tools* that would follow a cycle not dissimilar to general software development, as represented in figure 1.1;
- secondly and equally important, as it is implied by the title of their article and talk, *Run Your Research: On the Effectiveness of Lightweight Mechanization* the aim was that not only can a programmer create their own language, but they can also abstract features of existing languages and then use Redex to *verify* their work. Such "lightweight mechanized verification" can be used even for simple tasks such as LaTeX typesetting.

The demo in their talk was focused on nine ICFP 2009 articles where they discovered various errors, ranging from simple LaTeX typesetting and typos to false theorems which most likely were due to missing hypotheses (probably deemed obvious, but not explicitly stated).

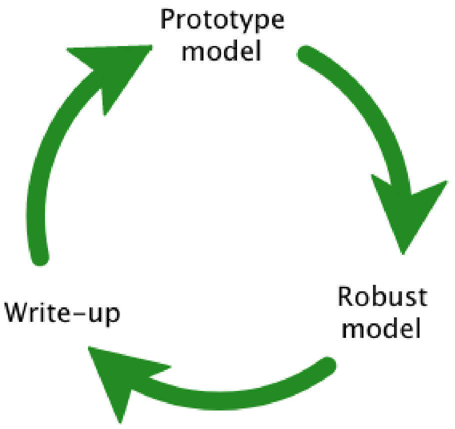


Figure 1.1: Semantics engineering cycle proposed in [FF+12]

In this dissertation, we will skip many of the theoretical aspects of Lisp and Scheme, stopping only at some details on `amb` and `call/cc` and assume the reader can catch most of the syntax on the go.

Without further ado, we will now skim through some features of Scheme and Racket that we will use throughout the rest of the dissertation. For further details, the standard references are [AS96] and [S+13] for Scheme and [rac20] and [FF14] for Racket.

1.1 Scheme and Racket crash course

Since both languages emerged from a common ancestor, Lisp, they use (as the name implies) a *list syntax*. Furthermore, most of the functions and predicates use a prefix notation and the mathematical operations use the so-called Polish (Łukasiewicz) notation. Some basic examples, including specific functions follow.

- `(+ (* 5 3) 1)` computes as $5 \cdot 3 + 1$;
- `(lambda (x) (rem x 2))` returns a function that computes the remainder of its argument modulo 2;
- `(cond (a instruction-a) (b instruction-b) (c instruction-c))` is a branching conditional expression. If `a` is true, then `instruction-a` is executed, else if `b` is true, `instruction-b` is executed and most of the times, `c` is taken to be `t` (true) such that the last branch is executed if everything else fails.

We should remark from the beginning that Racket is almost a superset of Scheme, and as such, its syntax contains that of Scheme. To be precise, Racket expands on the standards R5RS (1998) and R6RS (2009) of Scheme, diverging more significantly from R7RS (2013), but for the purpose of this dissertation, unless otherwise specified, all Scheme syntax presented is assumed to be valid Racket syntax.

While it is not typed in the modern sense of the word, Scheme differentiates between three basic types: *lists*, *functions* and *symbols*, the latter encompassing "everything else".

A specific syntax with its corresponding semantics is the *quoting mechanism*. As such, a *quoted expression*, denoted either by `(quote (expr))` or by `('(expr))` (another valid variation is `(#'(expr))`) is one that evaluates to itself, regardless of what `expr` contains. In brief, it turns `expr` into (a list of) symbols. Examples:

- `(lambda (x) '(+ 1 2 3))` is a function that associates to its argument the (literal) list `'(+ 1 2 3)` (notice that the sum is not evaluated, but it is returned literally);
- `(define animals '(cond cat dog mouse))` defines the variable `animals` to be the (literal) list `'(cond cat dog mouse)`.[2]

The opposite of quoting expressions is *unquoting*, which forces the evaluation and returns the final value in-place. The syntax for unquoting is a comma that precedes the expression that is evaluated. Examples:

- `(lambda (x) ,(+ 1 2 3))` is a function that returns 6 for all its arguments, since it associates to `x` the *final value* of `(+ 1 2 3)`, which is 6;

[2] We are aware that there's no "cond" animal, we included it to make it clear that it is not evaluated, i.e. the list is not seen as a branching conditional, but instead returned literally.

- `(define sum '(+ 1 2 3)) (define x (* ,sum 2))` makes `x` to be 12, since in its definition, the initially quoted `sum` is now forcibly evaluated.

In fact, the unquoting mechanism is an artifice of *metaprogramming*, more precisely an instance of a macro, which is one of the features that Lisps excel in.

Some more elements of syntax that will provide useful are listed briefly below.

- As seen above, functions and variables are defined with `(define ...)`. In fact, usually `define` is used for functions and variables are defined and given a value with `set`. There's also the variation `set!`, which forcibly overwrites whatever value the variable had;
- `cons` is the common function for appending to a list;
- `car` is the function that returns the head of the list and `cdr` returns its tail, so `(car '(1 2 3))` evaluates to 1 and `(cdr '(1 2 3))` evaluates to `(2 3)`;
- Comments are preceded by semicolons (`;`) and the convention is to use single semicolon for in-line comments and double semicolons for full-line comments. Also, in-line comments are used for writing the result of evaluating an expression, with an added arrow, e.g. `(+ 2 3) ; => 5`.

1.2 amb and call/cc

We will now give special attention to *ambiguous evaluation* using the function `amb` and also the *continuation* support included in Scheme, as they are specific features that will be used later on. Furthermore, from a historical perspective, Lisps were the first to support both of them. More languages followed, such as:

- `amb` implementation in many programming languages is comprehensively presented at [ros20];
- *continuations* are supported, for example, in Ruby's *Continuation* class ([rub20]), C++'s *context switching* ([Kow14]) and Haskell's `Control` monad ([has20]).

We will only present here the implementation of the two functions in Scheme, using mostly [AS96] and [non20]. Furthermore, the specifications for both are included in [S+13]. Since `amb` is implemented using `call/cc`, we will start with the latter.

By definition, the *continuation* of a computation is what will happen after the current computation is performed. For example, in the computation `(+ (* 2 3) 5)`, the continuation of the multiplication is the addition of 5 to the result. As such, one can abstract the continuation from this example with a lambda expression:

```
(define cont1
  (lambda (x) (+ x 5)))
```

Then, we get the same result as above by calling:

```
(cont1 (* 2 3))      ; => 11
```

While this example may not seem like much, the basic idea that one can capture "a part" of a computation and reuse it for some other purpose is important.

This concept is captured in the standard `call/cc` function in Scheme (sometimes implemented as `call-with-current-continuation`). Some basic examples are the following:

```
;; Nothing to capture
(define id              ; the identity function
  (lambda (k) k))
(call/cc id)            ; returns the function
                        ; since there's nothing left to do

;; Continuation is just the argument
(define apply-to-zero
  (lambda (k) (k 0)))
(call/cc apply-to-zero) ; returns 0
                        ; since that continues the function

;; Capture a further operation
(+ (call/cc
     (lambda (k) (k (* 1 2))))
      3)
;; is equivalent to
((lambda (v) (+ v 3)) (* 1 2))

;; Basically, the continuation only is (+ _ 3):
(lambda (x) (+ x 3))
```

The `call/cc` function can be used for saving a certain point of a computation. For example, we can save a backup copy of a function than can be used later:

```
(define z #f)                   ; initialize z with false
(+ (call/cc
     (lambda (k)
        (set! z k)              ; clear what's stored in z and put k
        (* 1 2)))
    3)                          ; => 5
```

In the example above, we have stored only the lambda-function in `z`, so even if the whole computation returns `5`, we have like a "bookmark" at the `lambda` definition which is stored in `z`. Henceforth, we can do:

```
(z 1)                ; => 4
```

The amb operator is a bit more complicated and although it is implemented in the standard library in some Scheme distributions, we will define it manually. We will follow the presentation in [non20]. This will define amb as a macro, which ensures that the arguments are not evaluated and as we will see, it is implemented using call/cc.

The basic idea, as outlined first in [McC61], is that the amb operator works as an ambiguous evaluation, with a variable number of arguments, in the following sense:

- if none of its arguments can be computed or if no arguments are given, it returns some kind of failure (we will use # f for false);

- if any (one or more) of its arguments can be computed, return all the possible values.

For example, McCarthy defines in pseudocode:

```
less(n) = amb(n-1, less(n-1)),
```

which is a "function" (quotes because it can return multiple values) that returns all the numbers that are less than the argument n. Thus, with n assumed to be a positive integer, amb tries to evaluate both n-1 and recursively compute less(n-1). If both can be computed, it returns both, which will happen until n = 1, when the final call will be amb(0, less(0)), that returns 0 and the computation stops, since none of the arguments won't work at the next iteration.

Here's how amb can be implemented in Scheme, using call/cc:

```
(define f #f)                                        ; 1

(define-syntax amb                                   ; 2
  (syntax-rules ()                                   ; 3
    ((_) (f))                                        ; 4
    ((_ a) a)                                        ; 5
    ((_ a b ...)                                     ; 6
      (let ((s f))                                   ; 7
        (call/cc                                     ; 8
          (lambda (k)                                ; 9
            (set! f (lambda ()                       ; 10
                          (set! f s)                 ; 11
                          (k (amb b ...))))          ; 12
          (k a)))))))                                ; 13
```

The basic ideas are the following:

- In the syntax-rules declaration, the underscore denotes the syntax (function) that is defined, amb;

- The first case (line 4) is when amb is called with no arguments, (amb), case in which it returns (f), which is false;

- Then, in line 5, if it is called with some argument, it will return that;
- If it is called with more than one argument (the ellipse being special syntax for zero or more arguments that could follow), then the actual work is done:
  - first, we introduce a variable `s` and bind it to `f`, which is `false`;
  - then, `call/cc` is called to capture the current continuation in `k`;
  - inside this `lambda`, `f` is reset and a recursive call is made, invoking `k` (the previous computation) to `amb` called with the rest of the arguments (starting with `b`);
  - finally, in line 13, the captured outermost `lambda` is applied to `a`, which makes the innermost `lambda` be called only after this computation, so first it tries to compute `(amb a)`

Remark 1.1: Note that by default, many Scheme and Racket implementations make `amb` behave step by step. That is, when called with multiple arguments, of which more than one can be computed, the first evaluation returns only the first computation. Calling `amb` again returns the second possible computation etc. Hence, for example, Racket 7.5 REPL, using the `#lang scheme` pragma, gives the following session, using the implementation above:

```
REPL> (amb "dog" "cat")
"dog"
REPL> (amb)
"cat"
REPL> (amb)
#f
```

Remark 1.2: One last word regarding the source code and the entire contents of this dissertation. While Racket comes with an exceptionally powerful and user-friendly IDE called `DrRacket` which is automatically installed when installing the language and for which the book [FF14] can be seen as an extensive user manual, we have chosen to write and test everything contained in this dissertation in GNU Emacs 26.3 running under Manjaro Linux. Scheme support is built in and for Racket, we used the full-featured REPL included in racket-mode.

CHAPTER 2

PLT REDEX

We will now detail the main topic of this thesis, that of the (PLT) Redex, as presented in [FF+12] and in extended form in [FF09].

For clarity, we will start with a simpler case, taken from the tutorial at [amb20] and also Part II of [FF09]. The full Redex will follow in a subsequent section.

2.1 The simple case

As mentioned in the introduction, Racket is a great toolbox for constructing languages, be them domain-specific (DSL), such as for systems programming, game engines, literate programming and more (as detailed in [rac20]). But what will interest us here are languages with a formal specification, namely those for which we will formulate grammars with production rules and reduction relations, which will allow us to check, for example, typing judgments as well as well-formedness of terms.

Redex is the module developed by the PLT team ([plt20]) and as such, all the snippets that we will be presenting should be evaluated in a source file that starts with the pragma announcing that we're using the full Racket language and the inclusion of the Redex module:

```
#lang racket
(require redex)
;; rest of the source code follows
```

We start by defining a simple language through its grammar, listing the non-terminals and their respective kinds. The syntax is quite self-explanatory, but we will include clarifying comments:

```
(define-language L                            ; the name of the language
  (e                                          ; non-terminals, which can be:
     (e e)                                   ; application
     (lambda (x t) e)                        ; lambda abstraction (t = type)
     x                                       ; variables
     (amb e ...)                             ; amb expressions
     number                                  ; numbers
     (+ e ...)                               ; sums
     (if0 e e e)                             ; nullity check (if-then-else)
     (fix e))                                ; fixed points
  (t (-> t t) num)                           ; functions between numeric types
  (x variable-not-otherwise-mentioned))      ; any other variable
```

Note that `t` is a type variable and hence the lambda abstraction is typed (commonly written as $\lambda(x : t).e$. Also note that the first six productions (application, λ-abstraction, variables, `amb` expressions, numbers and sums) are for general non-terminals (denoted in the source code by `e`), whereas the last two, namely the nullity checks and the fixed points are for the functional types `(t (-> t t) num)`, denoted, in functional pseudocode by `num -> num -> num`. The last clause is to allow the use of any variable that's not a keyword in the defined syntax, hence `x` can be anything but `e, lambda, t, amb` etc.

Now we can check whether a specific term can be obtained via this grammar. To differentiate it from a regular Racket expression, it must be preceded by the keyword `term`. So for example, we can check whether $\lambda x.x$ is a valid L-term by writing (either in the source code or in the REPL):

```
(redex-match                         ; do a pattern match
  L                                  ; in the language L
  e                                  ; see whether e can be
  (term (lambda (x) x)))             ; lambda x . x
;; => #f
```

We get `false`, since we used an untyped lambda expression, unlike the grammar which uses simply typed lambda calculus.

Another example which will work is:

```
(redex-match
  L
  e
  (term ((lambda (x num) (amb x 1))
            (+ 1 2))))
;; => (list (match (list (bind 'e '((lambda (x num) (amb x 1)) (+ 1 2))))))
```

So we were asking whether `e` can be the term we specified and since the answer is in the affirmative, we got a (one-term) list of all the bindings for the variables we asked for. Hence, the pattern-matching works with `e` being the whole term.

Of course, for multiple matches, Redex will return all the possibilities. A simple (and lengthy) example is the following:

```
(redex-match
  L
  (e_1 ... e_2 e_3 ...)
  (term ((+ 1 2)
         (+ 3 4)
         (+ 5 6))))
;; => (list
;;      (match
;;     (list
;;      (bind 'e_1 '())
;;      (bind 'e_2 '(+ 1 2))
;;      (bind 'e_3 '((+ 3 4) (+ 5 6)))))
;;    (match
;;     (list
;;      (bind 'e_1 '((+ 1 2)))
;;      (bind 'e_2 '(+ 3 4))
;;      (bind 'e_3 '((+ 5 6)))))
;;    (match
;;     (list
;;      (bind 'e_1 '((+ 1 2) (+ 3 4)))
;;      (bind 'e_2 '(+ 5 6))
;;      (bind 'e_3 '()))))
```

2.1.1 Extending with types and typing judgments

To properly support a typing system in the language we have defined, we will extend the language with typing judgments. As such, we will make it support expressions such as:

- $\Gamma \vdash x : t$, meaning that in the typing context Γ, the variable x is of type t;
- $y : t_1, \Gamma \vdash x : t_2$, meaning that x is of type t_2 in the context Γ which was extended with $y : t_1$.

add preliminaries for typing judgments & contexts?

Again, we present the syntax for extending the language and comment on its semantics.

```
(define-extended-language L+Gamma L
    [Gamma dot (x : t Gamma)])

(define-judgment-form
  L+Gamma
  #:mode (types I I O)
  #:contract (types Gamma e t)

  [(types Gamma e_1 (-> t_2 t_3))                       ; (1)
   (types Gamma e_2 t_2)
  --------------------------------
   (types Gamma (e_1 e_2) t_3)]

  [(types (x : t_1 Gamma) e t_2)                        ; (2)
  ------------------------------
   (types Gamma (lambda (x t_1) e) (-> t_1 t_2))]

  [(types Gamma e (-> (-> t_1 t_2) (-> t_1 t_2)))       ; (3)
  -----------------------------------------------
   (types Gamma (fix e) (-> t_1 t_2))]

  [--------------------------                           ; (4)
   (types (x : t Gamma) x t)]

  [(types Gamma x_1 t_1)                                ; (5)
   (side-condition (different x_1 x_2))
  --------------------------------------
  (types (x_2 : t_2 Gamma) x_1 t_1)]

  [(types Gamma e num) ...                              ; (6)
  ---------------------------
  (types Gamma (+ e ...) num)]

  [---------------------                                ; (7)
  (types Gamma number num)]

  [(types Gamma e_1 num)                                ; (8)
   (types Gamma e_2 t)
   (types Gamma e_3 t)
  -----------------------
  (types Gamma (if0 e_1 e_2 e_3) t)]

  [(types Gamma e num) ...                              ; (9)
  ------------------------------
  (types Gamma (amb e ...) num)])
```

Before commenting on the typing rules, we add the following remarks:

- Racket supports full Unicode symbols, so if the user prefers, they can use mathematical symbols in order to make the notation even clearer. We have preferred the ASCII set, but one can easily use Γ, $\longrightarrow$, λ etc. in Racket source code;
- Unlike most Scheme distributions, Racket encourages the use of square brackets to delimit "important" statements. Their use is not mandatory however and regular parentheses can be used throughout;
- The `#:mode` specifies that everything we will be declaring as syntax will follow the pattern `types Input Input Output`;
- The `#:contract` specifies that the syntax will contain the `types` keyword, then a typing context `Gamma`, then a non-terminal `e` (as in the initial definition of the language) and finally a type variable `t` (as output, cf. `#:mode`).

Then, the typing rules are written in a syntax that resembles the usual inference proof trees, with the hypotheses above the dashes (any more than or equal to 3 dashes are accepted) and the conclusion to follow. We will now explain the typing rules using a more "mathematical functional pseudocode", following the numbering in the source code.

(1) If in the context Γ we have $e_1 : t_2 \longrightarrow t_3$ and the argument $e_2 : t_2$, then the application $e_1 e_2 = e_1(e_2)$ will have type t_3. This is the basic rule for typing *functional evaluation;*

(2) If we extend the context Γ with some $x : t_1$ and in this extended context we have $e : t_2$, then the lambda abstraction that associates x to e, namely $\lambda(x : t_1).e$ will be a function $t_1 \longrightarrow t_2$. This is the basic rule for typing *lambda abstraction;*

(3) Given a higher-order function e in the context Γ, with type $e : (t_1 \longrightarrow t_2) \longrightarrow (t_1 \longrightarrow t_2)$, taking the fixed point of e will result in a function $t_1 \longrightarrow t_2$. This is the typing rule for *fixed points*;

(4) This is an axiom, usually called *extending contexts.* With no hypotheses, if we add the typing judgment $x : t$ to Γ, we get a context where $x : t$ holds true;

(5) Another approach to extending contexts is with a different variable. Thus, if we already have $\Gamma \vdash x_1 : t_1$ and we select some $x_2 \neq x_1$, then in the extended context $x_2 : t_2, \Gamma$ we still have $x_1 : t_1$;

(6) What follows is the *typing rule for addition.* Starting with variables of numeric type, their sum is still a numeric type;

(7) The special variable `number` is of type `num` and this is an axiom;

(8) We then have the typing rule for the *nullity check* (the simplest branching instruction we included in the language). Hence, starting with a numeric variable e_1 : num and some other variables e_2, e_3 of whatever types t, the instruction which can be read as

```
if (e_1 == 0) then e_2 else e_3
```

will have a type t, belonging to any of the branches;

(9) Finally, the *typing for amb*, but in a restricted situation, namely when it can evaluate a numeric type. Hence, if amb can evaluate some e : num, its output will be of type num, regardless of the rest of the arguments, since the evaluation returns the first successfull computation.

We should also add that if we decide not to use default Racket functions, Redex allows for the definition of *metafunctions* and as such, we can define the `different` metafunction, used in the side condition of rule 5:

```
(define-metafunction L+Gamma
  [(different x_1 x_1) #f]
  [(different x_1 x_2) #t])
```

Typing can now be tested as follows:

```
(judgment-holds
  (types dot                          ; we don't care about Gamma
    ((lambda (x num) amb x 1))
      (+ 1 2))
  t t)                                ; match for all types
;; => '(num)
```

The instruction checks whether the expression:

```
((lambda (x num) (amb x 1)) (+ 1 2))
```

can be typed using the grammar and the typing rules above. And if so, what can its type be. In this case, it returns `’(num)`, showing that the output of the expression will be of type num.

We can also extract only a part of the typing, not only for the whole expression, mentioning explicitly what we want:

```
(judgment-holds
  (types dot
    (lambda (f (-> num (-> num num)) (f (amb 1 2)))
      (-> t_1 t_2))                       ; it will be a function
    t_2)                                  ; give me only t_2 (range)
;; => '((-> num num))
```

Actual tests can also be written, such as:

```
(test-equal
  (judgment-holds
    (types dot (lambda (x num) x) t) t)
  (list (term (-> num num))))

(test-equal
  (judgment-holds
    (types dot (+ 1 2) t) t)
  (list (term (-> num num))))
;; => FAILED
;; actual: '(num)
;; expected: '((-> num num))
```

Note that by default, Racket does not output anything if the test succeeds and only reports an error if it fails. But we can summarize the tests with:

```
(test-results)
;; => 1 test failed (out of 2 total).
```

2.1.2 Extending with reduction relations

Now we add to our language reduction relations, which are defined by cases. As in the previous cases, we present the syntax, then explain its semantics.

```
(define-extended-language Ev L+Gamma
  (p (e ...))                          ; 1
  (P (e ... E e ...))                  ; 2
  (E (v E)                             ; 3
     (E e)                             ; 4
     (+ v ... E e ...)                 ; 5
     (if0 E e e)                       ; 6
     (fix E)                           ; 7
     hole)                             ; 8
  (v (lambda (x t) e)                  ; 9
     (fix v)                           ; 10
     number))                          ; 11
```

Hence, we are extending the language `L+Gamma` into what becomes `Ev`. At this step, we are extending it with non-terminals and contexts that will be the subject of the reduction relations that will follow. We are now adding:

- a non-terminal for *programs*, `p`, which is made of a list of expressions, so it contains at least some `e` and maybe some more;

- the P is also for programs, but enhanced with *evaluation contexts* (E, that follow), so such a non-terminal will include evaluation contexts for its expressions, `E e`;
- *E* denotes the *evaluation contexts*, which can appear:
  - with values v (e.g. quoted expressions, namely self-evaluating variables), case in which it does nothing, as it appears in postfix notation;
  - to provide evaluation contexts for expressions that are to be evaluated, thus appearing in prefix notation, `E e`;
  - in sums, nullity checks and fixed points, to provide contexts for the expressions;
  - as the special-matching variable `hole`, which matches ellipses.
- finally, *values* are allowed for evaluated lambda-terms, for fixed points and for numbers.

In the reduction relations we will also need to lift the regular sum function to work with Redex terms, so we define a meta-function in the extended language above:

```
(define-metafunction Ev
  Sigma : number ... -> number
  [(Sigma number ...)
   ,(apply + (term (number ...)))])
```

Note the use of the unquoted expression, which forces the evaluation, making the definition be smiliar to a Haskell `foldr`, sending the `+` inside the Redex `term` (in fact, the `foldr` function also exists in Racket and can be used alternatively in this case).

Now we can define the reduction relations[1]:

[1] To be precise, we will use a Redex function for substitution, `subst`, which should be included from the module `tut-subst`, as explained in [amb20], §1.4, where the implementation of the non-standard function is also explained. We have skipped this technicality, however, to streamline the presentation and also since its meaning we appreciate to be quite transparent.

```
(define red
  (reduction-relation
    Ev
    #:domain p
    (--> (in-hole P (if0 0 e_1 e_2))
         (in-hole P e_1)
         "if0true")
    (--> (in-hole P (if0 v e_1 e_2))
         (in-hole P e_2)
         (side-condition (not (equal? 0 (term v))))
         "if0false")
    (--> (in-hole P ((fix (lambda (x t) e)) v))
         (in-hole P (((lambda (x t) e) (fix (lambda (x t) e))) v))
         "fix")
    (--> (in-hole P ((lambda (x t) e) v))
         (in-hole P (subst x v e))
         "beta-reduction")
    (--> (in-hole P (+ number ...))
         (in-hole P (Sigma number ...))
         "summation")
    (--> (e_1 ... (in-hole E (amb e_2 ...)) e_3 ...)
         (e_1 ... (in-hole E e_2) ... e_3 ...)
         "amb")))
```

Therefore, we are defining a reduction relation that is called `red`, which applies to the language `Ev` and whose domain will be `p`, which was the non-terminal for programs. Then, the syntax has the general form

```
(--> (redex) (reduct) "name"),
```

with the special operator `-->`, followed by the reducible expression (also called a *redex*, its most reduced form (called a *reduct*) and an optional friendly name of the reduction relation which will prove useful a bit later.

Also note that all the reduction relations, except the one for `amb` are preceded by the keyword `in-hole` to mean that such expressions can have contexts. The omission of the keyword for `amb` is explained by the fact that in the extended language `Ev` we have no context for `amb`, which means that it is not extended, hence `amb` will only appear context-less, as it appeared in `L` initially and extended in `L+Gamma`.

As such, the `if0true` rule will look for programs in context `P`, where it finds a nullity check with the first argument to be precisely `0` and thus will return the true case. A similar reduction holds for the false case, where the "else" branch is returned.

The reduction for fixed points is given recursively and will be explained and visualized better below, when we typeset and render graphically `red`.

Next, we find the regular β-reduction for the (simply typed) λ-calculus, defined via substitutions. It can be written in a more conventional manner like so:

$$(\lambda(x\ :\ t).e)v \longrightarrow e[x/v]$$

Next we have the summation rule, which only specifies that `Sigma` should be used instead of the standard `+`.

Finally, the `amb` rule shows that if `amb` succeeds on the first argument, we just get that.

Writing tests for the reduction relation is simple and flexible. Redex provides two functions, one being `test-->>`, which tests the transitive closure of the `-->` relation we defined and the other is `test-->`, which only checks one step. Here are some examples:

```
(test-->> red
  (term ((if0 1 2 3)))
  (term (3)))

(test-->> red
  (term ((+ (amb 1 2) (amb 10 20))))
  (term (11 21 12 22)))

(test--> red
  (term ((+ (amb 1 2) 3)))
  (term ((+ 1 3) (+ 2 3))))

;; if multiple results are possible, all should be specified
(test--> red
  (term ((+ 1 2) (+ 3 4)))
  (term (3 (+ 3 4)))
  (term ((+ 1 2) 7)))

(test-results)  ; => 4 tests passed.
```

2.1.3 Graphically rendering and typesetting reductions

Finally, to end this example, we show how Redex provides very user-friendly tools to visualize the reduction steps and it can also typeset the relations, thus reaching the goal that Findler mentioned in [FF+12], to eliminate the typos in manually LaTeX typesetting such details.

One can typeset in-place the reduction relation in a very simple manner with the command:

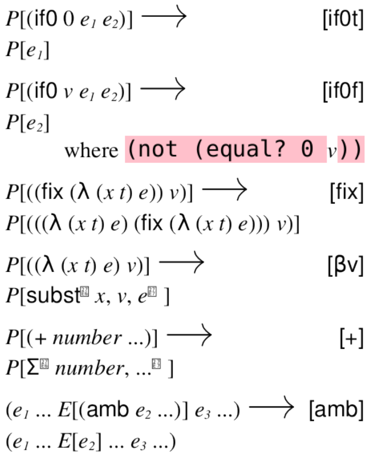
`(render-reduction-relation red)`,

which outputs an image. Note that both GUI Emacs and DrRacket support image rendering, but if you are using a terminal emulator with limited or no image support, the export solution which we detail below should be used.

In this case, we get the picture in figure 2.1.

To output the picture in a PostScript file, one can use the `pict` Racket library, which allows for further customizations that we will skip:

```
(require pict)
(scale (vl-append
        20
        (language->pict Ev)
        (redeuction-relation-> pict red))
        3/2)
;; => outputs the image to a .ps in the current directory
```



P[(if0 0 e_1 e_2)] $\longrightarrow$ [if0t]
P[e_1]

P[(if0 v e_1 e_2)] $\longrightarrow$ [if0f]
P[e_2]
where `(not (equal? 0` v`))`

P[((fix (λ (x t) e)) v)] $\longrightarrow$ [fix]
P[(((λ (x t) e) (fix (λ (x t) e))) v)]

P[((λ (x t) e) v)] $\longrightarrow$ [βv]
P[subst⟦ x, v, e⟧]

P[(+ $number$ ...)] $\longrightarrow$ [+]
P[Σ⟦ $number$, ...⟧]

(e_1 ... E[(amb e_2 ...)] e_3 ...) $\longrightarrow$ [amb]
(e_1 ... E[e_2] ... e_3 ...)

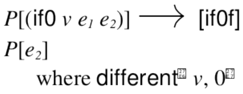
Figure 2.1: The initial rendering of the `red` reduction relation

Note that we have a highlighted function that Redex did not recognize properly, since it is a default Racket function. To eliminate this highlighting, we must turn the Racket function into a Redex metafunction. Or we can just use `different`, that is already a metafunction, defined earlier. Thus, the snippet below can be either used to replace the `if0false` rule in `red` or used and typeset per se:

```
(define if0-false-rule
  (reduction-relation Ev
    #:domain p
    (--> (in-hole P (if0 v e_1 e_2))
         (in-hole P e_2)
         (side-condition (term (different v 0)))
         "if0false")))

;; typeset it separately
(render-reduction-relation if0-false-rule)
```

which gives figure 2.2.



P[(if0 v e_1 e_2)] $\longrightarrow$ [if0f]
P[e_2]
where different⟦ v, 0⟧

Figure 2.2: Fixed pink highlight for `if0false` rule

Further tweaks can be made, e.g. to use more or better Unicode symbols, but we will skip them.

We close this section by mentioning `traces`, which is a great tool embedded in Redex that shows all the reduction paths for a certain term, according to the reduction relation specified. Its dependency is Graphviz, an open-source tool for interactive graph visualization. Thus, the call:

```
(traces red
  (term ((+ (amb 1 2)
            (amb 10 20)))))
```

shows all the reduction steps, with labels, to fully reduce the specified term. In this case, we get figure 2.3.

This figure was exported from Graphviz, which also provides some customization tools, along with a "Fully Reduce" button, which expands the figure until the final result, which is `(11 12 21 22)`. This is shown in figure 2.4 (notice the extra two bottom rows). The tool also reports the various reduction paths displayed, in this case (with the full reduction) the number being 26.

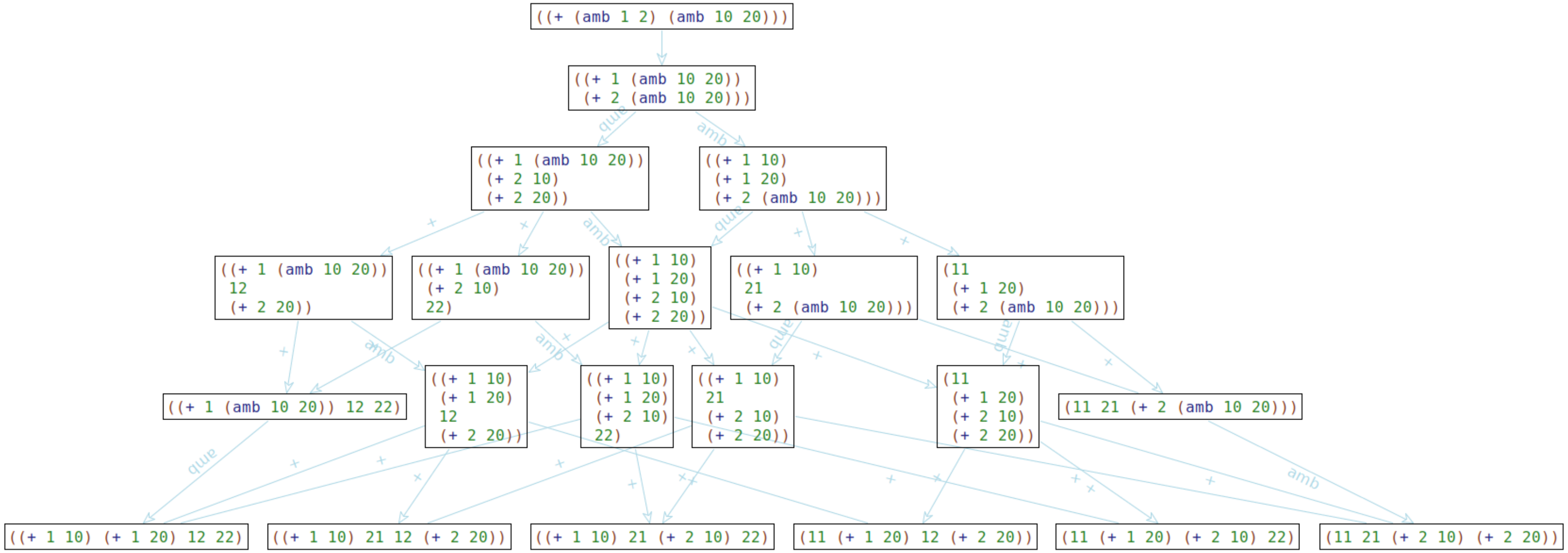


Figure 2.3: The incomplete reduction steps using `red` for `(term ((+ (amb 1 2) (amb 10 20))))`

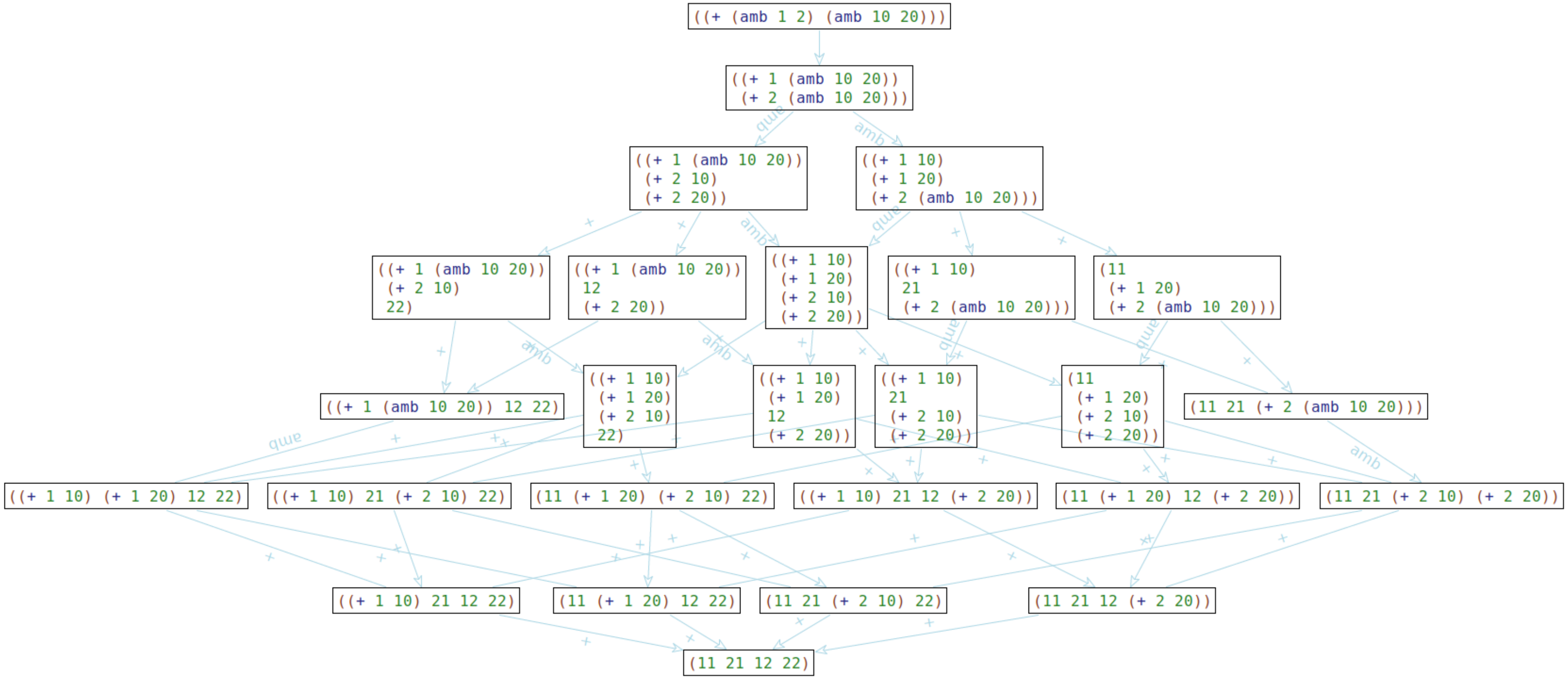


Figure 2.4: The full reduction steps using `red` for `(term ((+ (amb 1 2) (amb 10 20))))`

2.2 The POPL case

We now turn to the more “serious” use of Redex, namely that presented at POPL 2012 and detailed in [FF+12] and [FF09].

Having explained in detail the simple, but comprehensive example above, we now jump straight into the action. As such, we first define the simple language `Lambda-c`, encompassing basic (untyped) lambda calculus, plus `call/cc`, then extend it with evaluation contexts (and holes), plus the corresponding reduction relations.

So basically, we start with the following simple BNF grammar:

$$
\begin{aligned}
e \; ::= \; & (e \; e \; \dots) \\
& \mid x \\
& \mid (\lambda(x \dots).e) \\
& \mid \texttt{call/cc} \\
& \mid + \\
& \mid \texttt{number},
\end{aligned}
$$

which is readily written in Redex as:

```
#lang racket
(require redex)

(define-langauge Lambda-c
  (e (e e ...)
     x
     (lambda (x ...) e)
     call/cc
     +
     number)
  (x variable-not-otherwise-mentioned))
```

Then we extend it with contexts, formally written in BNF as:

$$
\begin{aligned}
e \; ::= \; & \dots \mid (A \; e) \\
v \; ::= \; & (\lambda(x \dots).e) \\
& \mid \texttt{call/cc} \\
& \mid + \\
& \mid \texttt{number} \\
E \; ::= \; & (v \dots E \; e \dots) \\
& \mid [\,]
\end{aligned}
$$

and then in Redex as:

```
(define-extended-language
  Lambda-c/red Lambda-c
    (e ... (A e))
    (v (lambda (x ...) e)
       call/cc
       +
       number)
    (E (v ... E e ...)
       hole))
```

Aside from the *evaluation context* E, we have now introduced the *abortion context* A, whose role can be seen in the reduction rules which follow below, first presented in a mathematical form, then in Redex:

- $E[(Ae)] \longrightarrow e$ (*abort*);
- $E[(\texttt{call/cc}\,v)] \longrightarrow E[(v(\lambda x.(AE[x])))]$, for x a *fresh variable* (*call/cc*);
- $E[((\lambda(x\,..._1).e)v\,..._1)] \longrightarrow E[e\{x := v, \dots\}]$ (*β-reduction*);
- $E[(+\texttt{number}\dots)] \longrightarrow E[\Sigma(number\dots)]$ (Σ).

Note that the rule for β-reduction used two kinds of ellipses, to show that after applying the rule, the rest of the computation could be different. Also, the `call/cc` reduction basically shows how the call with current continuation is equivalent to some lambda expression, in which the special context for **abortion** was used.

abort (exit) or continue??

Now in Redex:

```
(define red
  (reduction-relation
   Lambda-c/red
   #:domain e
   (--> (in-hole E (A e))
        e
        "abort")
   (--> (in-hole E (call/cc v))
        (in-hole E (v (lambda (x) (A (in-hole E x)))))
        (fresh x)
        "call/cc")
   (--> (in-hole E ((lambda (x ..._1) e) v ..._1))
        (in-hole E (subst e (x v) ...))
        "beta-reduction")
   (--> (in-hole E (+ number ...))
        (in-hole E (Sigma number ...))
        "sum")))
```

We assume that the metafunctions `Sigma` and `subst` are already defined, as in the previous section.

One more extension we can add is to allow the reductions to happen anywhere in redexes, since as defined in the extension `Lambda-c/red`, the first item of an expression in an `E`-context is a value `v`:

```
(define-extended-language
  anywhere-Lambda-c Lambda-c/red
  (E (e ... E e ...)
     hole))

;; extend the reductions by copying them
(define anywhere-red
  (extend-reduction-relation
    red anywhere-Lambda-c))
```

2.3 Extensive and Randomized Tests

Randomized testing in Redex is detailed in [KF09]. Also, as it is explained in [FF+12], Redex (and Racket) supports QuickCheck tests, in the style of the well-known combinator library initially developed in and for Haskell ([qui19]).

Before going into more details, we mention a simple test that can be run, using `redex-check`. The general syntax and more complex use is explained at [red20]. We will first use the form:

`(redex-check G n e),`

in which the boolean-valued expression `e` is seen as a predicate which is universally quantified over `n` and evaluates it for random terms that are generated from the non-terminal `n` belonging to the grammar `G`. Tweaks can be added, such as seeds for the random generation, trace printing, number of attempts, restrictions for the generated terms etc., but we will explore them later. First, a quick example:

```
(redex-check
  Lambda-c/red e
    (or (redex-match Lambda-c/red v (term e))   ; is it a term OR
        (cons?                                  ; is it a list
            (apply-reduction-relation red       ; after reducing ?
                (term e)))))
;; => counterexample found after 9 attempts: S
```

The example checks whether in the grammar `G` of the language `Lambda-c/red`, when quantifying over expressions `e`, what follows can be an expression `e`. Then, the disjunction (which must be a boolean-valued function) checks whether either `term e` is a value `v` in the language or after reducing it using `red` it is still a list.

The check reports that a free variable is a counterexample for the test, of course, since it's neither reducible, nor a value. This can be fixed by adding a special reduction relation to spot precisely free variables:

```
;; ... add to red ...
(--> (in-hole E x)
     error
     "free variable")
;; ...
```

We also add that [KF09] contains numerous excellent examples of faulty specifications or ugly bugs in the case of a language that is very similar to what we have been using in this section. For example, the fact that we have lifted the usual `(+)` to the metafunction `Sigma` is more important than it seems. This is because we have thus avoided the bug of checking whether `(+)` itself is a term, which is in principle allowed by the extension `Lambda-c/red`.

Back to the main topic of the section, that of randomized tests. Although it appears simplistic, the general syntax and options of `redex-check` allows for an impressive amount of flexibility and it is the main tool responsible for randomized and extensive tests in Redex. In [FF+12], the authors focused on claims that were central in some POPL papers, so they could run tests to check exactly those claims or theorems. We will try to do something similar in the next chapter, where we construct our own language to showcase some relevant design features, but currently, we will just detail a bit more on the options allowed by `redex-check`.

As detailed in [red20], the general syntax of this tool is the following:

```
  (redex-check template property-expr kw-arg ...)
   template     =     language pattern
         |     language pattern #:ad-hoc
         |     language pattern #:in-order
         |     language pattern #:uniform-at-random p-value
         |     language pattern #:enum bound
         |      language #:satisfying
             (judgment-form-id pattern ...)
         |      language #:satisfying
             (metafunction-id pattern ...) = pattern
   kw-arg     =     #:attempts attempts-expr
         |     #:source metafunction
         |     #:source relation-expr
         |     #:retries retries-expr
         |     #:print? print?-expr
```

```
    |      #:attempt-size attempt-size-expr
    |      #:prepare prepare-expr
    |      #:keep-going? keep-going?-expr
```

The general use is to search for counterexamples to `property-expr`, which is seen as a predicate that is universally quantified over the pattern variables that are found in `template`. How this works is that `redex-check` constructs and tests a candidate counterexample by choosing a random term `t` based on `template` and then evaluates `property-expr` using the match-bindings that are produced when matching `t` against `pattern`.

Most of the options are available for controlling how `t` is generated:

- `language pattern`: Redex uses an ad-hoc strategy for generating patterns that are used for tests. For the first 10 seconds, it will use in-order enumeration to pick terms, then it alternates between in-order enumeration and an ad-hoc random generator and after 10 minutes, it uses only the ad-hoc generator. The options `#:ad-hoc` and `#:in-order` forces it to use only the respective methods;

- the `#:uniform-at-random p-value` either selects a natural number at random, if the enumeration of the patterns is finite or it selects from a geometric distribution with probability specified by `p-value`. The range of this selection can be set with the `enum bound` option, which makes it check numbers only up to `bound`;

- the `#:satisfying` option filters test cases that only satisfy a particular judgment that is then specified and a similar filter can be used for metafunctions;

- `attempts-expr` controls how many random terms are generated and the size and complexity of the terms tends to increase after each failed attempt;

- the `#:prepare` keyword provides an optional function that can be used to modify the test cases, e.g. to apply a specific reduction relation to all of them or any other transformation via a `lambda` expression;

- the `source` keyword is followed by a metafunction or reduction relation which makes the test distribute evenly only with respect to that metafunction or reduction relation, i.e. it will generate only patterns that are similar to those found in the respective metafunction or relation.

CHAPTER 3

MY EXAMPLES

IDEA 1:

- See Typed Racket for how it handles types;
- Take some (reasonably sophisticated) examples of type theory from *Pierce, TAPL* (e.g. recursive types, subtyping);
- Implement them in Redex and verify stuff.

————————————

IDEA 2:

- See PML - *Intuitionistic Type Theory*;
- See its pragmatic part in *Programming in MLTT*;
- Write as much as possible of it in Redex/Racket and verify stuff.

————————————

IDEA 3:

- See ACL2 (e.g. *Kaufmann - Computer Aided Reasoning*);
- Write parts of it in Redex and verify stuff.

————————————

IDEA 4:

- Take a (reasonably simple) algebraic theory, such as category theory, group theory or even sheaf theory;

- Look for an Agda/Coq implementation;
- Translate it in Redex/Racket and verify stuff;
- The plus side: if it doesn't go well, I can comment why Redex is (not) suited as a proof assistant (also goes for the ACL2 idea).

————————————

IDEA 5:

- Take some chapters from the Program Verification course;
- Implement them in Redex;
- Verify examples;
- If it doesn't go well, I can comment why Redex is (not) suited as a program verification tool.

Am I missing essential stuff? Is the *reduction relations* part essential? My ideas focus most on judgments.

————————————

IDEA 6:
If reduction and rewriting is indeed the focus:

- Read *Baader, Nipkow – Term Rewriting and All That*, Chapter 8;
- Implement Gröbner Bases and the Buchberger Algorithm;
- Verify it.

BIBLIOGRAPHY

[amb20] Amb: A redex tutorial. https://docs.racket-lang.org/redex/tutorial.html, 2020. Accessed: March 2020.

[AS96] Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1996.

[FF09] Matthias Felleisen and Robert Findler. *Semantics Engineering with PLT Redex.* MIT, 2009.

[FF⁺12] Matthias Felleisen, Robert Findler, et al. Run your research: On the effectiveness of lightweight mechanization. *POPL*, 2012.

[FF14] Matthias Felleisen and Bruce Findler. *How to Design Programs.* MIT Press, 2014.

[Gro20] The PLT Group. Pyret. https://www.pyret.org/, 2020. Accessed: March 2020.

[has20] The control monad. http://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Cont.html, 2020. Accessed: March 2020.

[KF09] Casey Klein and Bruce Findler. Randomized testing in plt redex. In *Proc. Scheme and Functional Programming*, pages 26–36, 2009.

[Kow14] Oliver Kowalke. Context switching with call/cc. https://www.boost.org/doc/libs/1_72_0/libs/context/doc/html/context/cc.html, 2014. Accessed: March 2020.

[Lan64] Peter Landin. The mechanical evaluation of expressions. *Comput. J.*, 6:308–320, 1964.

[lon20] Long tutorial. https://docs.racket-lang.org/redex/redex2015.html, 2020. Accessed: March 2020.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Commun. ACM*, 3(4):184–195, April 1960.

[McC61] John McCarthy. A basis for a mathematical theory of computation. *Western Joint Computer Conference*, 1961.

[McC62] John McCarthy. Towards a mathematical science of computation. *IFIP-62*, 1962.

[non20] A gentle introduction to non-determinism in scheme. https://ebzzry.io/en/amb/, 2020. Accessed: March 2020.

[Plo75] Gordon Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[plt20] The PLT group. https://racket-lang.org/people.html, 2020. Accessed: March 2020.

[qui19] Quickcheck: Automatic testing of haskell programs. https://hackage.haskell.org/package/QuickCheck, 2019. Accessed: March 2020.

[rac20] Racket. https://racket-lang.org/, 2020. Accessed: March 2020.

[red20] redex-check documentation. https://docs.racket-lang.org/redex/The_Redex_Reference.html?q=redex-check#%28form._%28%28lib._redex%2Freduction-semantics..rkt%29._redex-check%29%29, 2020. Accessed: March 2020.

[ros20] Amb implementation. http://rosettacode.org/wiki/Amb, 2020. Accessed: March 2020.

[rub20] Continuation in ruby 2.5.0. https://ruby-doc.org/core-2.5.0/Continuation.html, 2020. Accessed: March 2020.

[S$^+$13] Gerald Sussman et al. Revised7 Report on the Algorithmic Language Scheme. Technical report, Scheme Working Group, 7 2013.