A Tutorial on Using PLT Redex for Mechanizing Access Control Models

Philip W. L. Fong
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
pwlfong@ucalgary.ca

January 17, 2018

1 Introduction

We are only human, and humans make mistake. This applies particularly to those of us who formalize access control models as state transition systems. The language we use include set theoretic notations and automata theoretic constructs. This is not the language we speak on a daily basis. Formal modelling is an intellectually challenging task, and even seasoned researchers make mistakes from time to time.

One way to reduce errors is to employ a formal methods tool to help "mechanize" the modelling process [2, 13]. These tools help check the mathematics of the users, and assist in discovering "bugs" in their models. Think of these tools as debuggers for mathematical models. Not only that, the resulting models are typically executable. This allows the researcher to test run her model, to see if it is doing what she is expecting it to do. Some of these tools also provide visualization facilities for users to explore the state space of an access control system, thereby granting them deeper insight of their work.

In this tutorial, we will look at one such tool — PLT Redex [5, 10]. PLT Redex is originally designed as a semantics engineering tool. Its target users are programming language designers who would like to formalize the operational semantics of a programming language. They usually do so by formally defining an abstract state machine that captures the run-time behaviour of programs in the language being modelled. It is also a relatively lightweight formal methods tool, allowing its users to quickly come up with an executable semantic model. For this latter reason, we borrow this tool for modelling access control systems.

Preliminaries. PLT Redex is embedded in Racket, a dialect of the Scheme programming language. A prerequisite of using PLT Redex is therefore learning to program in Scheme. Fortunately, it is not a difficult language if you have prior exposure to functional programming. Some higher education institutes actually use Scheme in their first-year introductory courses for programming

```
1   ;;;
2   ;;; Beginning of Source File
3   ;;;
4
5   #lang scheme
6   (require redex)
```

Figure 1: Language specification and library import.

```
193 ;;;
194 ;;; End of Source File
195 ;;;
196
197 (module+ test
198   (test-results))
```

Figure 2: Report test statistics.

[1, 6]. A good entry point to the language is to read Chapter 2 and Sections 3.1–3.2 of TSPL4 [4], which is available online at `www.scheme.com/tspl4/`:

> R. Kent Dybvig. *The Scheme Programming Language (4th Edition).* The MIT Press, 2009.

(Familiarity of Chapter 6 will also be helpful.)

PLT Redex is available on the Racket platform, which can be downloaded at the following site:

`http://racket-lang.org`

The platform is also good for learning Scheme.

In this tutorial, we will use the Graham-Denning Model [8] for discretionary access control as an example. Specifically, we will give a mechanized specification of the Graham-Denning Model as formulated by Li and Tripunitara [12, §4.1]. Familiarity with that formulation is a prerequisite of this tutorial.

2 Patterns and Terms

This tutorial comes with a source file `GD.rtk`. In the following, we will go through the file line by line to explain how PLT Redex can be employed to specify the Graham-Denning Model. The source code lines are displayed in figures, such as those in Figures 1 and 2. The lines are numbered for easy reference in the main text.

```
(define-language GD
   ⋮
)
```

Figure 3: Defining a new language GD.

Let us first make a few comments about the general structure of a model specification in PLT Redex. Figure 1 are typical lines that begin a specification. It tells the Redex environment what language dialect the following source code is written in. The Racket environment recognizes several dialects, of which `scheme` is one (line 5). Then on line 6 we import the module that provides the Redex features.

Figure 2 lists typical lines at the end of a specification. A major benefit of mechanizing an access control model in Redex is that one gets to "test" the formulation to see if it is doing what one is expecting it to do. In the following, we will set up quite a number of tests for our code. The lines in Figure 2 print a brief report of the test results.

Exercise 1 *Start up the DrRacket environment, and load the source file `GD.rtk`. Press `Run` to execute the definitions and the test code. Do the test cases pass?*

Make sure you run the source file, or else the definitions required for the following steps would not have loaded, and you would not be able to follow along the tutorial.

Figure 3 shows the skeleton of a language definition. As we shall see in §3, this construct specifies a context-free grammar which introduces a number of non-terminal symbols that we will use for pattern matching, a feature that sits at the foundation of PLT Redex. Understanding how pattern matching works is the key to understanding the rest of Redex. This is the topic for the rest of this section. For now, all you need is to take note that `GD` is the name of the language we introduce through `define-language`.

2.1 Terms

Redex is embedded in Scheme. One could imagine using Scheme to "implement" the Graham-Denning Model, but such a specification would be overly procedural, and thus too cryptic for comprehension. Redex essentially provides a layer of notations that makes the specification of state transition systems not only convenient, but also highly declarative. What makes the specification convenient and declarative is the use of terms and patterns.

At the end of the day, we would like to use Scheme data structures for representing states, and so state transitions are but transformations that take as input Scheme data values representing states and produce as output Scheme data values representing states. To facilitate the specification of such transformations, states are specified in Redex terms. **Redex terms are essentially convenient notations for specifying Scheme data values.**

Consider the following examples:

```
(term (1 2 3))
⇝ (1 2 3)
```

The construct `term` allows one to switch from Scheme to the Redex term notation, so that one can specify a Scheme data value not in Scheme, but in the Redex term notation. The Scheme data value specified in the Redex term notation is then returned as the value of the `term` construct. The above example is does not do anything grand. All it does is to "escape" to the term notation, so we can write down the list `(1 2 3)`. The corresponding Scheme data value is produced as the result of evaluation. The reader probably notice that in the Redex term notation, data values do not need to be quoted, as in Scheme:

```
'(1 2 3)
⤳ (1 2 3)
```

Many Redex facilities expect operands written as Redex terms. In those situations, one does not even need to enclose the term in the `(term ...)` construct.

By itself, the Redex term notation does not seem to offer much convenience except for eliminating the need for quoting. The benefits of using term notation will become obvious once we learn to couple it with pattern matching.

2.2 Pattern Matching

Patterns are specifications for expressing constraints over the type and structure of data. A pattern can be "matched" against a term. Here are two simple examples:

```
(redex-match? GD natural 1)
⤳ #true

(redex-match? GD natural #true)
⤳ #false
```

The `redex-match?` construct checks if a given term (e.g., `1`) matches a given pattern `natural`. The construct is used in the context of a language specified using `define-language`. Here we specify `GD` as the language (Figure 3). Our examples in this section are independent of the grammar of `GD`. We supply it only because we need to provide a language name for `redex-match?`. Returning to the above examples, the Redex pattern language supports a number of built-in types. For example, the type `natural` refers to the type of natural numbers (i.e., 0, 1, 2, ...). Other built-in patterns include `integer` and `boolean`. When such type names are used as a pattern, it matches against a term that denotes a value of that type. That is why the first match above succeeds, but the second fails.

Not only can patterns specify the type of data, they can also be used for specifying the structure of data. The following are examples:

```
(redex-match? GD (natural_1 natural_2 natural_3) (term (3 7 4)))
⤳ #true

(redex-match? GD (natural_1 natural_2 natural_3) (term (1 2)))
⤳ #false
```

Here the pattern specifies a list of length 3, with all the members being natural numbers. So the term `(3 7 4)` matches the pattern, but the term `(1 2)` does not. Note that the three natural numbers are given distinct names in the pattern: `natural_1`, `natural_2`, and `natural_3`. Each name, or more precisely, each ***pattern variable***, starts with a type (`natural`), followed by an underscore ("_"), and then an index (`1`). These pattern variables are significant because pattern matching produces bindings for these variables, as the following example demonstrates:

```
(redex-match GD (natural_1 natural_2 natural_3) (term (3 7 4)))
⇝ (
    #(struct:match
       (#(struct:bind natural_1 3)
        #(struct:bind natural_2 7)
        #(struct:bind natural_3 4))
   )
```

Here, we use a different Redex construct, `redex-match`, which not only checks if the match is successful, but also construct a `match` structure that contains bindings to the pattern variables that are produced by the match. For instance, here `natural_1` binds to the number `3`, `natural_2` binds to the number `7`, and `natural_3` binds to the number `4`.

One can also specify patterns that match a sequence of objects with the same type:

```
(redex-match GD (natural_1 natural_2 ...) (term (3 7 4)))
⇝ (
    #(struct:match
       (#(struct:bind natural_1 3)
        #(struct:bind natural_2 (7 4))))
   )
```

While the pattern `natural_1` matches a single natural number, the pattern `natural_2 ...` matches a (possibly empty) sequence of natural numbers. When the pattern `(natural_1 natural_2 ...)` is matched against a non-empty list (e.g., `(3 7 4)`), `natural_1` will match the first element of the list (i.e., `3`), and `natural_2 ...` will match the rest of the list (i.e., `7`, `4`). As for the bindings, the pattern variable `natural_2` will bind to a list containing the matched sequence of terms (i.e., `(7 4)`).

In the examples we have seen so far, pattern matching generates only one match (i.e., one set of bindings). That is not true in general. It is possible that the pattern allows multiple matches against a given term. That is why the return value of `redex-match` is a *list* of `match` structures, as in the following example:

```
(redex-match GD (natural_1 ... natural_2 natural_3 ...) (term (3 7 4)))
⇝ (
    #(struct:match
       (#(struct:bind natural_1 ())
        #(struct:bind natural_2 3)
        #(struct:bind natural_3 (7 4))))
    #(struct:match
       (#(struct:bind natural_1 (3))
        #(struct:bind natural_2 7)
        #(struct:bind natural_3 (4))))
    #(struct:match
       (#(struct:bind natural_1 (3 7))
        #(struct:bind natural_2 4)
        #(struct:bind natural_3 ()))))
     )
```

There are three possible matches in the example above, depending on whether to match `natural_2` with `3`, `7`, `4`. The patterns `natural_1 ...` and `natural_3 ...` essentially match with the elements to the left and right of `natural_2`.

Exercise 2 *What matches will the following expression produce? Try to write them down on a piece of paper before you try typing the expression below into DrRacket.*

```
(redex-match
  GD
  (natural_1 ... natural_2 natural_3 ... natural_4 natural_5 ...)
  (term (3 7 4)))
```

Actually, the `...` construct can be used for augmenting any arbitrary pattern.

```
(redex-match GD ((natural_1 natural_2) ...)
                (term ((1 2) (3 4) (5 6))))
⇝ (
    #(struct:match
      (#(struct:bind natural_1 (1 3 5))
       #(struct:bind natural_2 (2 4 6))))
   )
```

The pattern `(natural_1 natural_2) ...` specifies a sequence of terms, each of which has the form `(natural_1 natural_2)`. That is, the pattern specifies a sequence of pairs. The pattern variable `natural_1` binds to the list of natural numbers appearing as the first members of the pairs, while `natural_2` binds to the second members.

2.3 Constructing Terms with the Help of Pattern Matching

Equipped with pattern matching, the Redex term notation becomes a very convenient tool for constructing Scheme values.

```
(redex-let GD
           ([(natural_1 natural_2 ...)
             (term (1 2 3 4))])
           (term (natural_2 ...  natural_1)))
⇝ (2 3 4 1)
```

The `redex-let` construct has a syntax analogous to `let`. The difference is that between the square brackets we specify patterns and expressions (rather than variables and expressions). Pattern matching will occur, and the pattern variable bindings are collected. Then in the body of the `redex-let`, the pattern variable bindings can be used in the term notation. Here, `(natural_2 ... natural_1)` constructs the a list that begins with the sequence signified by `natural_2 ...`, followed by `natural_1`. The effect is "rotating" the first element of `(1 2 3 4)` to the end of the list.

Consider another example:

```
(redex-let GD
           ([((natural_1 natural_2) ...)
             (term ((1 2) (3 4) (5 6)))])
           (term (natural_1 ... natural_2 ...)))
⇝ (1 3 5 2 4 6)
```

The sequences captured by pattern variables `natural_1` and `natural_2` are now listed alongside one another in the output term.

Here is yet another example:

```
(redex-let GD
           ([((natural_1 natural_2) ...)
             (term ((1 2) (3 4) (5 6)))])
           (term ((natural_2 natural_1) ...)))
⇝ ((2 1) (4 3) (6 5))
```

The output term is still a list of pairs, but the order of the two elements in a pair is now reversed.

While the term notation is highly expressive, it does not mean that it can do everything that you want. There are times you would want to switch back to Scheme and just express your thoughts in Scheme. Here is an example:

```
(redex-let GD
           ([(natural_1 natural_2 natural_3 ...)
             (term (1 2 3 4))])
           (term (natural_1 ,(+ 10 (term natural_2)) natural_3 ...)))
⇝ (1 12 3 4)
```

The pattern variables `natural_1`, `natural_2` and `natural_3` will bind to `1`, `2` and the sequence `3`, `4` respectively. The list constructed as output begins with `natural_1` (and thus the resulting list begins with `1`), and ends with `natural_3 ...` (and thus the resulting list ends with `3` and `4`). What is interesting is the second member of the resulting list. The comma (“`,`”) let us escape from the term notation back to the Scheme programming language. So the expression `(+ 10 (term natural_2))` is interpreted as Scheme code. The expression adds 10 to `(term natural_2)`. Here, within Scheme, we escape back to the term notation, so as to retrieve the binding of the pattern variable `natural_2`. The second member of the resulting list, `12` is the result of adding `10` to the value of the pattern variable `natural_2`. This sort of notation switching is not particularly readable, and thus it should be avoided if possible. PLT Redex provides metafunctions for exactly this purpose, but that feature is beyond the scope of a minimalist tutorial.

When you use Redex to specify a state transition system, it is unlikely you will need to use constructs such as `redex-match?`, `redex-match`, or `redex-let`. They are used in this section only to illustrate how pattern matching works. But the general idea will be the same. Pattern matching will be employed to check the structure of a term as well as to extract the components of that term, and then pattern variables are substituted in the specification of another term.

Exercise 3 *There are lots of features about terms and patterns that we are not able to examine in a short tutorial like this one. Please spend some time browsing through the §4.1 and §4.2 of the Redex reference manual [7], to find out the full syntax of the term notation and the pattern*

```
7   (define-language GD
8     [Sub    (sub natural)]
9     [PObj   (obj natural)]
10    [Obj    Sub
11            PObj]
12    [AR     own
13            control]
14    [BR     (variable-except own control)]
15    [TR     (trans BR)]
16    [Right  BR
17            AR
18            TR]
19    [Priv   (Sub Right Obj)]
20    [S      (Sub ...)]
21    [O      (PObj ...)]
22    [R      (BR ...)]
23    [M      (Priv ...)]
24    [State  (st natural natural S O R M)]
25  )
```

Figure 4: Definition of protection states.

language. In general this is how you learn PLT Redex: not only by reading this tutorial, but also by going to the reference manual from time to time to find out more. You do not need to understand everything in the reference manual in one sitting. Just browse through §4.1 and §4.2 quickly to have an idea of what features are offered. When you build your own model, you may recall something you have seen in the reference manual, that is when you want to do a more in depth reading before you try out the feature.

From now on, this advice will not be repeated. It is assumed that when we discuss `define-language`, `reduction-relation`, etc, you will go to the reference manual to do a little bit of browsing.

3 Protection State

We are now ready to examine the `GD` language as specified in Figure 4.

3.1 The Basic Entities

Lines 8–18 define the basic entities in our formulation of the Graham-Denning Model. The lines within a `define-language` are essentially context-free grammar rules defining non-terminals. For example, lines 8–18 can be read as the following context-free grammar rules:

Sub ::= `(sub` *natural*`)`

PObj ::= `(obj` *natural*`)`
Obj ::= *Sub* | *PObj*
AR ::= `own` | `control`
BR ::= *variable*
TR ::= `(trans` *BR*`)`
Right ::= *BR* | *AR* | *TR*

Non-terminal symbols are displayed in italics, and terminal symbols are displayed in typewriter fonts. It is not hard to recognize that, with a `define-language`, each pair of square brackets delimits the definition of a non-terminal symbol. The first symbol appearing next to the open square bracket is the non-terminal being defined. Each term following that defines an alternative production for the non-terminal. We go through each of the non-terminal symbols in turn.

1. The non-terminal `Sub` signifies a subject (line 8). It is represented by a list of length two, beginning with the symbol `sub` (a tag to make explicit the type of the subject when it is printed), followed by a numeric identifier that uniquely identifies the subject. Essentially, anything that is specified on the right-hand side is considered concrete syntax (i.e., terminal), unless it is a non-terminal. In our case, `natural` is a built-in pattern (natural number). It is used here as a non-terminal signifying the set of natural numbers.

2. The non-terminal `PObj` denotes a ***proper object*** (line 9). Recall that every subject is considered an object. An object that is not also a subject is called a proper object. A proper object has a structure analogous to that of a subject.

3. The non-terminal `Obj` denotes an object, which can be either a subject or a proper object (lines 10 and 11).

4. The non-terminal `AR` signifies an ***administrative right***, namely, either `own` or `control` (lines 12 and 13).

5. The non-terminal `BR` denotes the basic rights (line 14). We could have defined `BR` as follows:

   ```
   [BR   variable]
   ```

   The built-in pattern `variable` denotes the set of all Scheme symbols.[1] This would allow us to use symbols such as `read` or `write` as basic access rights. Yet we want to be careful to exclude `own` and `control` from being basic rights, so we instead use the pattern `(variable-except own control)` to make sure that the match will be successful only if the Scheme symbol is not an administrative right.

6. The non-terminal `TR` denotes the transferrable forms of the basic rights (line 15). For example, if `read` is a basic right, then `(trans read)` is the transferrable version of `read`. Note that only basic rights can be made transferrable. Administrative rights do not have transferrable counterparts.

[1]Recall that PLT Redex is originally designed for modelling the operational semantics of programming languages. Scheme symbols are typically used for representing variables in the object language being formalized via Redex. That is why Scheme symbols are called "variables."

7. The non-terminal `Right` signifies access rights, which can be basic rights, administrative rights, or transferrable rights (lines 16–18).

Non-terminal symbols can be used as patterns during pattern matching. This means that by introducing new non-terminals, we are essentially equipping the Redex pattern matching mechanism with more patterns to work with.

```
(redex-match? GD (Sub Right Obj)
                 (term ((sub 2) (trans read) (obj 3))))
⇝ #true

(redex-match GD (Obj_1 ... Obj_2 Obj_3 ...)
                (term ((sub 1) (obj 2) (obj 3))))
⇝ (
    #(struct:match
      (#(struct:bind Obj_1 ())
       #(struct:bind Obj_2 (sub 1))
       #(struct:bind Obj_3 ((obj 2) (obj 3)))))
    #(struct:match
      (#(struct:bind Obj_1 ((sub 1)))
       #(struct:bind Obj_2 (obj 2))
       #(struct:bind Obj_3 ((obj 3)))))
    #(struct:match
      (#(struct:bind Obj_1 ((sub 1) (obj 2)))
       #(struct:bind Obj_2 (obj 3))
       #(struct:bind Obj_3 ())))
   )
```

3.2 The Structure of a Protection State

We now turn our attention to lines 19–24 of Figure 4, which describe the structure of a protection state. In particular, a protection state is defined in line 24, through the definition of the non-terminal `State`.

```
[State   (st natural natural S O R M)]
```

A state is a list of fixed length, beginning with the symbol `st` (to explicitly indicate its type when the state is printed), which is in turn followed by 6 components:

1. The first component is a natural number, which we will call the ***subject counter***. The number essentially signifies the range of subject identifiers that are considered legitimate. For example, if the counter is 2, then `(sub 0)` and `(sub 1)` are legitimate subjects, but `(sub 2)` and `(sub 3)` are not. This counter is tracked in the protection state so that when a new subject is created, there is a deterministic way for us to assign a fresh subject identifier for it: essentially the current counter value is the next unused subject identifier. The counter gets incremented after the new subject is created. This is a slight deviation from the formulation of Li and Tripunitara. In their formulation, the new subject to be created can be non-deterministically selected from any inactive subject. In a mechanized formulation,

we eliminate non-determinism to render the state space smaller, more amenable to human examination using visualization tools.

2. The second component, which we will call the ***proper object counter***, is again a natural number. This counter marks the range of legitimate identifiers for proper objects.

3. The third component is an `S`, which is defined on line 20 as:

   ```
   [S    (Sub ...)]
   ```

   So the component is simply a list of subjects. This ***subject list*** holds the set of all active subjects: subjects that have been created but not yet destroyed. Even though a subject may be legitimate according to the current value of the subject counter, that subject may have been destroyed from the state, and thus the subject will not appear in the subject list.

   We assume that the list is always sorted in strictly ascending order.

4. The fourth component has type `O` (line 21):

   ```
   [O    (Obj ...)]
   ```

   This ***proper object list*** contains the set of all proper objects that are still active in the state (i.e., not yet destroyed). It is assumed that the list is sorted in strictly ascending order.

5. The fifth component is of type `R`, which is defined on line 22 as follows:

   ```
   [R    (BR ...)]
   ```

   This **basic rights list** contains the list of all basic rights supported by this Graham-Denning system.

6. The **access control matrix** is defined on line 23 as follows:

   ```
   [M    (Priv ...)]
   ```

   That is, the access control matrix is represented by a list of granted privileges. Each privilege has the following structure (line 19):

   ```
   [Priv    (Sub Right Obj)]
   ```

   Recall that a `Right` can be a basic right (`BR`), and administrative right (`control` or `own`), or the transferrable version of a basic right.

   We assume that this list is always sorted in strictly ascending order. We will say more about this ordering in §3.4.

Note that we have made the assumption that the lists `S`, `O` and `M` are sorted in strictly ascending order. These components are supposed to be representing sets in the mathematical formulation. By requiring that the list representations to be sorted, we essentially impose a canonical representation for sets: i.e., a set is canonically represented by a list sorted in strictly ascending order. While a set can in principle be represented by many different lists (due to multiple ordering or repeated membership), there is a unique canonical representation as a list sorted in strictly ascending order. Canonicity is imposed so that later on, when the visualization tools are employed for depicting the state space, it will be able to recognize the equivalence of states, and thus draw a smaller state space that is more amenable for manual, human examination.

3.3 Preliminary Testing

Figure 5 lists the entities we construct for testing purposes. In this Graham-Denning system, there are only two basic rights: `read` and `write`.

Note that state `st2` (and its corresponding access control matrix `m2`) is supposed to be obtained from state `s1` (access control matrix `m1`) by subject `s1` creating a new object `o2`. Note that `s1` is marked as the owner of `o2` in the access control matrix. Note also the increment of the proper object counter.

State `st3` (access control matrix `m3`) is obtained from `st1` by subject `s0` transferring the basic right `r1` (for object `o0`) to subject `s1`. Note that `s0` possesses the transferrable version of `r1`, but the right received by `s1` is not transferrable.

State `st4` (access control matrix `m4`) is obtained from `st2` by subject `s0` creating a new object `o3`. Note again the marking of ownership in the resulting access control matrix as well as the increment of the proper object counter.

As we will be using these constructed entities for further testing, it is important to make sure that we have formulated them properly: i.e., in accordance to the syntax specified in Figure 4. I cannot stress enough about the importance of this assurance. It is extremely easy for a model specifier to create very sophisticated test cases, only to find out later that the test cases are actually formulated incorrectly. And the most common error in the formulation of test cases is that the test cases do not even conform to the structure mandated by the model (i.e., `GD` in our case). We therefore formulate some further tests to type-check the constructed entities (Figure 6).

3.4 Looking Ahead: How will a State Change

As mentioned above, the access control matrix is represented as a list of privileges that are sorted in strictly ascending order. When we are to formulate our state transition relation, we will then have to maintain this order. Specifically, we will need to be able to insert a new privilege into a sorted list while maintaining the sorted order.

We implement `insert-priv` for the mentioned purpose (Figure 7). It employs two helper functions. The function `right-compare` in Figure 8 compares two access rights (`Right`). A non-transferrable right (i.e., a basic right or an administrative right) is considered "smaller" than (ordered before) a transferrable right. Among the basic rights and administrative rights, they are ordered according to their string representation. Figure 9 lists the helper function `priv-comp`, which compares two privileges (`(Sub Right Obj)`). The function first attempts to order the two privileges by their subject identifiers. If the subject identifiers are the same, then the function

```
(define s0 (term (sub 0)))
(define s1 (term (sub 1)))

(define o0 (term (obj 0)))
(define o1 (term (obj 1)))
(define o2 (term (obj 2)))
(define o3 (term (obj 3)))

(define r1 (term read))
(define r2 (term write))

(define br (term (,r1 ,r2)))

(define m1
  (term ((,s0 (trans ,r1) ,o0) (,s1 ,r2 ,o1))))
(define m2
  (term ((,s0 (trans ,r1) ,o0) (,s1 ,r2 ,o1) (,s1 own ,o2))))
(define m3
  (term ((,s0 (trans ,r1) ,o0) (,s1 ,r1 ,o0) (,s1 ,r2, o1))))
(define m4
  (term ((,s0 (trans ,r1) ,o0) (,s0 own ,o3) (,s1 ,r2 ,o1)
         (,s1 own ,o2))))

(define st1
  (term (st 2 2 (,s0 ,s1) (,o0 ,o1) ,br ,m1)))

(define st2
  (term (st 2 3 (,s0 ,s1) (,o0 ,o1 ,o2) ,br ,m2)))

(define st3
  (term (st 2 2 (,s0 ,s1) (,o0, o1) ,br ,m3)))

(define st4
  (term (st 2 4 (,s0 ,s1) (,o0 ,o1 ,o2 ,o3) ,br ,m4)))
```

Figure 5: Examples of various entities.

```
(module+ test
  (test-equal (redex-match? GD Sub s0) #true)
  (test-equal (redex-match? GD Sub s1) #true)
  (test-equal (redex-match? GD Obj o0) #true)
  (test-equal (redex-match? GD Obj o1) #true)
  (test-equal (redex-match? GD Obj o2) #true)
  (test-equal (redex-match? GD Obj o3) #true)
  (test-equal (redex-match? GD BR r1) #true)
  (test-equal (redex-match? GD BR r2) #true)
  (test-equal (redex-match? GD M m1) #true)
  (test-equal (redex-match? GD M m2) #true)
  (test-equal (redex-match? GD M m3) #true)
  (test-equal (redex-match? GD M m4) #true)
  (test-equal (redex-match? GD State st1) #true)
  (test-equal (redex-match? GD State st2) #true)
  (test-equal (redex-match? GD State st3) #true)
  (test-equal (redex-match? GD State st4) #true)
)
```

Figure 6: Test cases for ensuring syntax conformity.

```
(define (insert-priv priv matrix)
  (if (null? matrix)
      (list priv)
      (case (priv-comp priv (car matrix))
        [(-1) (cons priv matrix)]
        [( 0) matrix]
        [(+1) (cons (car matrix)
                    (insert-priv priv (cdr matrix)))])))
```

Figure 7: Inserting a privilege into an access control matrix.

```
(define (right-comp r1 r2)
  (if (symbol? r1)
    (if (symbol? r2)
        (let ([s1 (symbol->string r1)]
              [s2 (symbol->string r2)])
          (cond
            [(string<? s1 s2) -1]
            [(string>? s1 s2) +1]
            [else             0]))
        -1)
    (if (symbol? r2)
      +1
      (right-comp (second r1) (second r2)))))
```

Figure 8: Comparing two access rights.

```
(define (priv-comp priv1 priv2)
  (let ([s1 (first priv1)]
        [r1 (second priv1)]
        [o1 (third priv1)]
        [s2 (first priv2)]
        [r2 (second priv2)]
        [o2 (third priv2)])
    (cond
      [(< (second s1) (second s2)) -1]
      [(> (second s1) (second s2)) +1]
      [(and
        (eqv? (first o1) 'sub)
        (not (eqv? (first o2) 'sub))) -1]
      [(and
        (not (eqv? (first o1) 'sub))
        (eqv? (first o2) 'sub)) +1]
      [(< (second o1) (second o2)) -1]
      [(> (second o1) (second o2)) +1]
      [else (right-comp r1 r2)])))
```

Figure 9: Comparing two privileges.

```
118 (module+ test
119   (test-equal (insert-priv (list s1 r1 o0) m1) m3)
120   (test-equal (insert-priv (list s1 'own o2) m1) m2)
121   (test-equal (insert-priv (list s0 'own o3) m2) m4)
122 )
```

Figure 10: Test cases for `insert-priv`.

attempts to order the two privileges by their object identifiers. If both the subject and object identifiers are the same, then the function orders the two privileges by their access rights, using the helper function `right-comp`.

To ensure that `insert-priv` is implemented properly, we formulate a number of test cases in Figure 10. Specifically, the relationship among the four access control matrices `m1`, `m2`, `m3` and `m4` are used for checking that `insert-priv` behaves as expected.

4 State Transition

Figure 11 demonstrates how the state transition rules of an access control system can be specified as a ***reduction relation*** in Redex. Specifically, two of the 13 Graham-Denning rules as presented in [12, Figure 1] are captured here. A reduction relation is created (line 124) for the language `GD` (line 125). The reduction relation is stored in the Scheme variable `red` (line 123).

The first rule is specified in the production `((--> ...))` on lines 126–131. It is intended to capture the command $\textsf{transfer\_}r(i, s, o)$ of Li and Tripunitara. The second rule is specified in the production on lines 132–142. This rule captures the command $\textsf{create\_object}(i, o)$. We will examine these lines in greater details.

4.1 Object Creation

Since the second rule is easier to understand, let us examine it first. Li and Tripunitara defines the semantics of $\textsf{create\_object}(i, o)$ as follows:

```
**command** $\textsf{create\_object}(i, o)$
  **if** $o \notin O_\gamma \wedge i \in S_\gamma \wedge o \in \mathcal{O} \setminus \mathcal{S}$ **then**
    $O_{\gamma'} \leftarrow O_\gamma \cup \{o\}$
    $M_{\gamma'}[i, o] \leftarrow \textsf{own}$
```

In short, the object being created (o) must be a fresh proper object ($o \notin O_\gamma$ and $o \in \mathcal{O} \setminus \mathcal{S}$), and the initiator ($i$) will become the owner of the created object ($M_{\gamma'}[i, o] \leftarrow \textsf{own}$).

Our production in lines 132–142 captures the meaning of the command by specifying that, a protection state of the form below:

```
(st natural_1 natural_2 S (PObj ...) R M_1)
```

```
(define red
  (reduction-relation
    GD
    (--> (st natural_1 natural_2 S O R M_1)
         (st natural_1 natural_2 S O R M_2)
         (where (Priv_1 ... (Sub_1 (trans BR) Obj) Priv_2 ...) M_1)
         (where (Sub_3 ...  Sub_2 Sub_4 ...)  S)
         (where M_2 ,(insert-priv (term (Sub_2 BR Obj)) (term M_1)))
         (computed-name (term (transfer BR Sub_1 Sub_2 Obj))))
    (--> (st natural_1 natural_2
             S (PObj ...)
             R M_1)
         (st natural_1 ,(+ 1 (term natural_2))
             S (PObj ... (obj natural_2))
             R M_2)
         (where (Sub_1 ... Sub_2 Sub_3 ...) S)
         (where M_2 ,(insert-priv (term (Sub_2 own (obj natural_2)))
                                  (term M_1)))
         (computed-name
           (term (create-object Sub_2 (obj natural_2)))))
  )
)
```

Figure 11: Definition of transition relation.

will transition to a protection state of the following form:

```
(st natural_1 ,(+ 1 (term natural_2))
                                S (PObj ... (obj natural_2)) R M_2)
```

Note that the components `natural_1`, `S` and `R` remain the same after the transition. Three components, however, are altered.

1. First, the proper object counter `natural_2` is changed to `,(+ 1 (term natural_2))`. As we are creating a new proper object. The proper object counter should be incremented. The "`,`" escapes to the Scheme programming language, so we can compute `(+ 1 ...)`. The second argument to the integer addition, however, requires us to switch back to the Redex term notation so that substitution will allow us to retrieve the value of the pattern variable `natural_2`.

2. The second change concerns the list of proper objects `(PObj ...)`, which is replaced by `(PObj ... (obj natural_2))` in the resulting state. The effect is that the proper object list is extended by a new member at the end of the list. That member has `natural_2` as its identifier.

3. The third change concerns the access control matrix (i.e., the list of granted privileges), which changes from `M_1` to `M_2`. The new access control matrix `M_2` is computed in two steps, via the two `where` clauses on lines 138 and 139. A `where` clause has the general form:

   `(where` *pattern term*`)`

   It is typically used in a production to set up the bindings of pattern variables. Its effect is to match the Redex term *term* against the Redex pattern *pattern*. When there are multiple matches, the production will generate multiple states as results, one for each possible match. The first `where` clause (line 138) will non-deterministically selects a subject in the subject list `S` to be the initiator of the object creation operation. It does so by matching the subject list `S` against the pattern `(Sub_1 ... Sub_2 Sub_3 ...)`. The pattern variable `Sub_2` will match against each of the members of `S` in turn, thereby accounting for all possible initiators. Then the second `where` clause (line 139) will construct the new access control matrix `M_2`, by invoking the Scheme function `insert-priv` (thus the prefix "`,`" is employed to escape to Scheme). The first argument to `insert-priv` is the privilege specified by the term `(Sub_2 own (obj natural_2))`, indicating that the initiator of object creation is the owner of the newly created object. The second argument `M_1` is the access control matrix to which the new privilege is to be inserted. Notice that as we need Redex to substitute actual values into the pattern variables (e.g., `Sub_2`, `natural_2`, and `M_2`), we employ `(term ...)` to switch from Scheme back into Redex.

A last remark concerns the label of the production. In some cases, a production has a fixed name, and so the production name can be specified as a string that goes after the `where` clauses. In our case, however, we would like the name to clearly identify the initiator as well as the created object (as in $\mathsf{create\_object}(i, o)$). We therefore employ a `computed-name` clause to construct a name for the production (line 142). The general form of a `computed-name` clause is the following:

```
(computed-name scheme-expression)
```

The Scheme expression *scheme-expression* will be evaluated and then converted to a string. And that string will be used for identifying how the result of the production is generated. In our case, the name depends on the choice of `Sub_2` (the initiator) and `(obj natural_2)` (the newly created object).

4.2 Transfer of Basic Rights

Let us now return to the production on lines 126–131, which is intended to capture the transfer_$r(i, s, o)$ command of Li and Tripunitara, who endows it with the following semantics:

```
command transfer_$r(i, s, o)$
  if $r^* \in M_\gamma[i, o] \wedge s \in S_\gamma$ then
    $M_{\gamma'}[s, o] \leftarrow M_\gamma[s, o] \cup \{r\}$
```

In short, a subject (i) who possesses the transferrable version of a basic right for an object ($r^* \in M_\gamma[i, o]$) can initiate the transfer. The result is that an existing subject ($s \in S_\gamma$) will receive that right ($M_{\gamma'}[s, o] \leftarrow M_\gamma[s, o] \cup \{r\}$).

By examining lines 126 and 127, we notice that the only component of the protection state that is altered by this production is the access control matrix (changing from `M_1` to `M_2`). In the third `where` clause on line 130, `M_2` is obtained from inserting a new privilege into `M_1` (via a call to the Scheme function `insert-priv`). That privilege is the Redex term `(Sub_2 BR Obj)`. Here, `Sub_2` is the subject receiving the access right, `BR` is the basic right that is being transferred, and `Obj` is the object protected by the right.

The question is, how are the initiator, the receiver, the basic right and the object get generated. The answer is found in the where clauses of lines 129 and 130. The `where` clause on line 130 non-deterministically selects a privilege of the form `(Sub_1 (trans BR) Obj)` from the original access control matrix (`M_1`)). The pattern variable `Sub_1` is the initiator of the transition. The pattern `(trans BR)` ensures that the initiator has the transferrable version of the access right `BR` over the object `Obj`. The `where` clause on line 129 non-deterministically selects a subject `Sub_2` from the subject list `S` to be the receiver of the transfer. These choices are explicitly documented in the name constructed in the `computed-name` clause on line 131.

4.3 Reflections

From the discussion above, we can identify several important lessons in using PLT Redex for modelling state transition systems:

1. When we read a formulation such as that of Li and Tripunitara, the perspective is that a command is instantiated with actual arguments to produce a transition. Here the perspective is that the transition rules must generate all possible transitions. In your own modelling, make sure your formulation is able to exhaustively generate all possible transitions.

```
145 (module+ test
146   (test-->>E #:steps 1 red st1 st3)
147   (test-->>E #:steps 1 red st1 st2)
148   (test-->>E #:steps 1 red st2 st4)
149   (test-->>E #:steps 2 red st1 st4)
150 )
```

Figure 12: Test cases for the state transition relation.

2. We sometimes need to impose a canonical representation of states in order to help PLT Redex recognize equivalent states.[2] Canonicity is typically required when one is working with sets or other data types in which their Scheme representations are more fine grained than the original data types.

3. Often in the modelling of access control systems, we have to work with universe of entities with a countably infinite cardinality (e.g., the universe of subject identifiers, the universe of proper object identifiers). We often need to non-deterministically select inactive elements from these universes, and add them into the protection state as a result of entity creation (e.g., subject creation). To reduce the size of the state space, it is helpful to manage entity creation using a deterministic mechanism (e.g., using a counter to determine the next available identifier).

4.4 Testing the Transition Relation

A primary goal of mechanizing a state transition relation in PLT Redex is so that we can "run" the transition to see if it is doing what we expect it to do. In Figure 12 we employ `test-->>E` to do exactly this. Line 147 checks if `st2` is reachable from `st1` in one transition step. Similarly, line 149 tests if `st4` is reachable from `st1` in two transition steps.

4.5 Debugging the Transition Relation

Crafting good test cases is a good start of understanding our formulations. But an even more useful tool is to use the stepper actually "run" the state transition system step by step, like one would use a debugger to step through the individual lines of a program. The stepper can be invoked as follows in the DrRacket prompt:

```
(stepper reduction-relation state)
```

The argument *reduction-relation* is the reduction relation one would like to examine, and *state* is the initial state to start the examination. When the command is issued, a visual tool will be displayed. The user can choose a state from the screen (initially there is only one), and request

[2]This can actually be achieved via other means, but imposing canonicity is probably the most convenient way to do this.

the tool to apply the transition relation to the selected state. All the next states are generated and depicted on screen. The user can therefore examine the effect of a transition step in details, and then pick another state for the next transition step. This tool is particularly helpful for understanding the behaviour of the transition relation, and instrumental in assisting the model specifier in uncovering unexpected behaviour.

Exercise 4 *Invoke the stepper in the DrRacket prompt:*

```
(stepper red st1)
```

Interact with the tool to expand the part of the state space that you would like to explore. Carefully compare what you see with what you think the transition rules are doing.

4.6 Visualizing the State Space

Another very helpful tool is the tracer, which visually depicts the state space in graphical form. It can be invoked at the DrRacket prompt with arguments similar to the stepper:

```
(traces reduction-relation state)
```

The tracer is helpful because it simulates the effects of successive transitions, maps out an entire region of the state space, and connects the individual states with transitions that are labelled with names. This is very helpful for examining the topology of the state space.

The tracer will usually generate only a certain number of states, and then give the user an option for generating more. What is also useful is to request the tracer to not be too ambitious in its state generation, or else the depicted state space will be too crowded to be useful. This can be achieved by using the `reduction-steps-cutoff` command prior to invoking the tracer. For example, the following command will print the number of transition steps the tracer will simulate, starting from the initial state.

```
(reduction-steps-cutoff)
```

This number may be too big for a complex transition relation. It can be adjusted to a smaller number by the following command:

```
(reduction-steps-cutoff new-cutoff)
```

Of course, this needs to be done prior to the invocation of the tracer.

Exercise 5 *Set the cutoff to 3, and then invoke the tracer to examine the state space surrounding the state* `st1`*. Move the states around to help you examine the transitions.*

4.7 The Rest of the Transition Relation

The following exercise is the crown of this tutorial:

Exercise 6 *Formulate the rest of the Graham-Denning transition relation. That is, formulate the rest of the 13 rules that are presented by Li and Tripunitara [12, Figure 1].*

Do this incrementally. Formulate one rule. Then create a good number of test cases. Use the stepper and tracer to help you understand your own work.

```
151 (define (well-formed-state? state)
152   (let ([sub-count (second state)]
153         [obj-count (third state)]
154         [sub-list  (fourth state)]
155         [obj-list  (fifth state)]
156         [priv-list (seventh state)])
157     (and (well-formed-so-list? sub-count sub-list)
158          (sorted-so-list? sub-list)
159          (well-formed-so-list? obj-count obj-list)
160          (sorted-so-list? obj-list)
161          (well-formed-priv-list? sub-list obj-list priv-list)
162          (sorted-priv-list? priv-list))))
```

Figure 13: A state invariant for ensuring that states are well-formed.

5 State Invariants

A very helpful feature of PLT Redex is that the visualization tool `traces` can be used for testing if the transition rules preserve a certain state invariant. Of course testing does not replace theorem proving — whether the transition rules indeed preserve the state invariant still needs to be proved formally. Such testing, however, offers the model specifier rapid feedback on whether the formulation of the rules are obviously flawed.

We formulate an invariant that captures the notion that a state is ***well-formed***. The invariant is encoded as a scheme predicate (i.e., a function that returns a boolean value), `well-formed-state?`, as shown in Figure 13. Given a state `(st` n_1 n_2 `S O R M)`, the invariant asserts that the following six conditions hold:

1. Every subject `(sub` n`)` in the subject list `S` must have $n < n_1$. This condition is checked on line 157 of Figure 13 via a call to `well-formed-so-list?` (defined in Figure 14).

2. The subject indices in the list `S` are sorted in strictly ascending order. Implicitly this implies that there is no repetition in the list `S`. This condition is tested on line 158 of Figure 13 through an invocation of `sorted-so-list?` (see Figure 14 for its definition).

3. Every object `(obj` n`)` in the proper object list `O` must have $n < n_2$. Again, this check is performed by an invocation to `well-formed-so-list?` (line 159 of Figure 13).

4. The proper object indices in `O` are sorted in strictly ascending order. Thus `O` does not contain repeated entries. This condition is checked on line 160 of Figure 13, by a call to `sorted-so- list?`.

5. Every privilege `(Sub BR Obj)` in the access control matrix `M` must refer only to subjects in `S` and objects in either `S` or `O`. This condition is tested on line 161 of Figure 13, through an invocation of function `well-formed-priv-list?` (which is defined in Figure 14).

6. The privileges in list `M` are sorted in strictly ascending order (according to the definition of `priv-comp` defined above). Again, every privilege occurs at most once in `M`. This check is performed by a call to `sorted-priv-list?` on line 162 of Figure 13. The helper function `sorted-priv-list?` is defined in Figure 14.

Such an invariant is useful in two ways. The first use is to make sure that the protection states we create for testing are actually well-formed. Test case crafting is by itself an error-prone endeavor. If the test cases are incorrect, then they are useless. Figure 15 shows how the well-formedness invariant can be assuring the integrity of the test states.

A second use of the invariant is to test if the transition rules actually preserve well-formedness. That is, we want to find out if a transition can turn a well-formed protection state into an ill-formed protection state. We can do so by using the visualization tool `traces`.

```
(traces red st1 #:pred well-formed-state?)
```

By supplying `well-formed-state?` predicate as an extra argument to the tracer, the visualization tool will check each generated state for compliance to the invariant. Any state that does not satisfy the invariant will be highlighted in pink, thereby allowing the model specifier to detect bugs in the formulation of the transition rules.

Exercise 7 *On line 4 of Figure 11, the newly created proper object is placed at the end of the proper object list, thereby maintaining the sorted order of the list:*

```
(PObj ... (obj natural_2))
```

Replace the above term by the following:

```
((obj natural_2) PObj ...)
```

Now the newly created proper object is placed at the front of the proper object list. This formulation of object creation will produce a protection state that violates Condition 4 of the well-formedness invariant (i.e., the object list shall be sorted). Use `traces` to examine the state space, and provide the well-formedness invariant as an extra argument. Do you notice the ill-formed states produced by the new transition rule?

In general, when you formulate an access control model, you should try to think hard about what state invariants you want to maintain in the model. Then carefully specify them to help you check if the transition rules are formulated in such a way that the invariants are preserved.

Exercise 8 *Li and Tripunitara [12, §4.1] specify 7 state invariants that are supposed to be preserved by the Graham-Denning transition rules. Formulate them as Scheme predicates. Then use them to (a) debug the protection states you create for testing purpose, and (b) check if the Graham-Denning transition rules you have formulated (or the ones provided in Figure 11) indeed preserve these invariants.*

```
(define (well-formed-so-list? so-count so-list)
  (or (null? so-list)
      (and (< (second (first so-list)) so-count)
           (well-formed-so-list? so-count (rest so-list)))))

(define (sorted-so-list? so-list)
  (or (null? so-list)
      (null? (rest so-list))
      (and (< (second (first so-list))
              (second (second so-list)))
           (sorted-so-list? (rest so-list)))))

(define (well-formed-priv-list? sub-list obj-list priv-list)
  (or (null? priv-list)
      (let* ([priv (first priv-list)]
             [sub (first priv)]
             [obj (third priv)])
        (and (member sub sub-list)
             (or (member obj sub-list)
                 (member obj obj-list))
             (well-formed-priv-list? sub-list obj-list
                                     (rest priv-list))))))

(define (sorted-priv-list? priv-list)
  (or (null? priv-list)
      (null? (rest priv-list))
      (let ([priv1 (first priv-list)]
            [priv2 (second priv-list)])
        (and (< (priv-comp priv1 priv2) 0)
             (sorted-priv-list? (rest priv-list))))))
```

Figure 14: Helper functions for implementing the state invariant in Figure 13.

```
(module+ test
    (test-equal (well-formed-state? st1) #true)
    (test-equal (well-formed-state? st2) #true)
    (test-equal (well-formed-state? st3) #true)
    (test-equal (well-formed-state? st4) #true)
)
```

Figure 15: Ensuring that the protection states created for testing purpose are all well-formed.

6 What Next?

This tutorial is a minimalist one. There are lots of PLT Redex features that we have not looked at. We particularly want to encourage the adventurous readers to learn about metafunctions and judgments [7]. They will reduce the amount of procedural Scheme code that you have to mix into your supposedly declarative specification.

At the end, the best way to learn PLT Redex is to use the tool in your next modelling project. Try using Redex to formalize the Chinese Wall Model [3], the HRU Model [9], the Bell-LaPadula Model [11], or whatever access control model you are working with. That is the best way to gain experience with Redex.

References

[1] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 2nd edition, 1996.

[2] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLMARK challenge. In Joe Hurd and Tom Melham, editors, *Theorem Proving in Higher Order Logics*, volume 3603 of *LNCS*, pages 50–65. Springer, 2005.

[3] David F. C. Brewer and Michael J. Nash. The Chinese Wall security policy. In *Proceedings of the IEEE Symposium on Research in Security and Privacy (S&P'1989)*, pages 206–214, Oakland, California, USA, May 1989.

[4] R. Kent Dybvig. *The Scheme Programming Language*. The MIT Press, 4th edition, 2009.

[5] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009.

[6] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs*. The MIT Press, 2nd edition, 2014.

[7] Robert Bruce Findler, Casey Klein, Burke Fetscher, and Matthias Felleisen. *Redex: Practical Semantics Engineering (v 6.6)*. `http://docs.racket-lang.org/redex/`.

[8] G. Scott Graham and Peter J. Denning. Protection: principles and practice. In *Proceedings of the 1972 AFIPS Spring Joint Computer Conference*, volume 40, pages 417–429, Alantic City, New Jersey, USA, May 1972.

[9] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[10] 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 mechanism. In *Proceedings of the*

39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'2012), pages 285–296, Philadelphia, PA, USA, January 2012.

[11] Leonard J. LaPadula and D. Elliot Bell. MITRE technical report 2547, volume II: Secure computer systems. *Journal of Computer Security*, 4(2–3):239–263, 1996.

[12] Ninghui Li and Mahesh V. Tripunitara. On safety in discretionary access control. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 96–109, Oakland, CA, USA, May 2005.

[13] Benjamin C. Pierce, Peter Sewell, Stephanie Weirich, and Steve Zdancewic. It is time to mechanize programming language metatheory. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, volume 4171 of *LNCS*, pages 26–30. Springer, 2008.