Optimized Implementation of Faceted Execution in Racket

Lisa Hou

Advisor: Dave Wonnacott

March 26, 2020
Contents

0.1 Introduction .................................................. 1
0.2 Background & Motivation .................................... 2
0.3 Literature Review .............................................. 6
0.4 Methodology ...................................................... 11
0.5 Results & Future Work ......................................... 14
Abstract

This project examines issues in the domain of faceted execution, a way of equipping programming languages with support for privacy policies. I extend RACETS (Micinski et al., 2018), an implementation of faceted execution in Racket, by attempting to provide support for user-defined macros.
0.1 Introduction

From social media applications like Twitter to the healthcare portal hosted by your medical service provider, access restrictions can be found everywhere in modern software. Unfortunately, implementing these access restrictions, or "privacy policies", in code can be very difficult, as the programmer must constantly perform conditional checks. When a significant portion of the code ends up handling sensitive information such as in likes of banking and healthcare, errors can lead to serious consequences (Polikarpova, 2018).

Policy-agnostic programming is a programming paradigm that transfers the responsibility of handling privacy policies from the programmer to the programming language itself. After the initial privacy policies are set up, the rest of the code can be written naively, while remaining secure. One example of policy-agnostic programming is faceted execution (Yang, 2013). Faceted execution is all about manipulating data structures called facets. A facet is a tuple of the form \( \langle l \?, v_H : v_L \rangle \) where \( l \) is a label, \( v_H \) is the high-confidentiality value, and \( v_L \) is the low-confidentiality value (Micinski et al., 2018). The actual value of the data to be protected is placed in \( v_H \), and some default, public value like 0 or null is placed in \( v_L \). \( v_H \) can only be accessed by observers that satisfy the policy identified by the facet’s label; other observers can see only \( v_L \).

Among the various strategies computer scientists have utilized to implement faceted execution, one method is to integrate faceted execution into an
existing language. The RACETS programming language by Micinski et al. (2018) adopts this strategy by integrating the Racket language with macros. In this project I present an improved implementation of RACETS.

## 0.2 Background & Motivation

RACETS represents privacy policies as predicates which take an argument key and return true or false to indicate permission to view a sensitive value. They are declared with the let-label form:

```scheme
(define my-policy
  (let-label l (lambda (x) (equal? x "Alice")) l))
```

The code above creates a policy and binds it to the name my-policy. The policy specifies that only users identifying themselves as “Alice” may view the sensitive value of any facet protected by the policy. A faceted data value is created with the fac form:

```scheme
(define my-facet (fac my-policy 1 0))
```

my-facet is defined with Alice’s policy, the high-confidentiality value 1, and the low-confidentiality value 0. The obs form is used to view the value of a facet. The first argument to obs is the policy. The second argument is a token to pass to the policy predicate, which is often a string identifying the role of the entity observing the facet. The third argument is the facet itself.

```scheme
(obs my-policy "Alice" my-facet)
```
The expression above will evaluate to 1, as the argument ”Alice” satisfies the facet’s policy. By contrast, the expression below will evaluate to 0, since ”Anna” does not satisfy the facet’s policy.

```scheme
1 (obs my-policy "Anna" my-facet)
```

The policy passed to obs must match the policy that the facet was created with in order to view the sensitive data. If the policies don’t match, the default value will be shown.

Programmers use the let-label, fac, and obs forms to create and observe facets. Policies are created with let-label, sensitive data is enclosed with fac, and sensitive data are disclosed using obs. The mechanisms behind RACETS allows the rest of program to treat facets like normal values, which includes applying normal functions.

Suppose that a normal, policy-naive add3 function was defined as follows:

```scheme
1 (define (add3 x) (+ 3 x))
```

Without the special mechanisms behind RACETS (add3 (fac p 4 0)) would throw an error, + expects a number not a facet. However, the mechanisms in RACETS implicitly transforms the syntax to (fac p (add3 4) (add3 0)), distributing the function application across the two branches of the facet. This transformation, which applies to all function applications in RACETS, ensures that functions like add3 can be applied to faceted values even when written for non-faceted values.

Faceted values may also be nested to support more complex privacy policies. Nested faceted values are constructed just as are with regular values –
except that instead of a regular value, another faceted value (fac your-policy ...) would be in its place. Furthermore, any Racket data structure may be turned into a faceted value – a facet would simply encapsulate two different versions of the data structure, one on each branch.

Given that macros are central to the implementation of RACETS, we present a mini crash course on them here, based on Greg Hendershott’s “Fear of Macros” guide. Unlike C macros, which are extremely limited in their functionality, Racket macros are much more sophisticated. In Racket, macros are syntax-transforming procedures that run at compile-time. They allow programmers to define new syntactic structures without having to modify the language’s parser. At compile time, the Racket compiler drives a process called macro-expansion, where all the macros in the program are transformed, and then substituted into the original code.

To define a function we use define, but to define a macro we use define-syntax. Macros are similar to functions, except instead of taking concrete values as arguments, they take a syntax object (analogous to a syntax tree) and return a syntax object. The following macro uses the syntax constructor for syntax objects to return a new syntactic form, ignoring its input:

```
(define-syntax (hello-world stx)
  (syntax (displayln "hello world!")))
```

A macro is invoked just like a regular function:

```
(hello-world (- 4 5))
```

Note that the input, (- 4 5) is ignored.
Since macros basically take syntax trees, the stx parameter in the call to say-hello above is bound to the entire syntax object of (hello-world (- 4 5)), and not the concrete value -1. We can convert syntax objects to and from lists for easier manipulation, using syntax→datum and datum→syntax. This is useful for when you want to do things like reversing arguments:

```
(define-syntax (reverse-syntax stx)
  (datum→syntax stx
   (reverse (cdr (syntax→datum stx))))))
(reverse-syntax 4 5 -)
```

The result of the call on the last line is 1.

Since stx is bound to the entire syntax tree of the macro call, (syntax→datum stx) returns (reverse-syntax 4 5 -) and not '(4 5 -). Consequently, we must call cdr on this list before reversing it, to remove the name of the macro.

Since macros are expanded at compile-time, they are capable of things that are impossible for regular functions. my-and, defined below, defines the short-circuiting boolean and function, so that if a in (my-and a b) evaluates to false, then b is never evaluated. If this were instead a function, b would still be evaluated, because arguments to functions are always evaluated before they are passed.

```
(define-syntax (my-and stx)
  (syntax-case stx ()
    [(_ a b)
      #'(if a b #f)]))
```
The result of the call on the last line is 1.

my-and uses a pattern-matching function called syntax-case. Each identifier in the (a b) pattern clause matches a single top-level expression (which may be an atomic value or a compound structure like a list). The underscore matches the name of the macro itself, my-and, and the symbols a and b match arguments to the macro.

Finally, Racket macros are hygienic, which means that identifiers introduced in the macro are automatically protected from conflicting with identical symbols in the original source code.

0.3 Literature Review

In this section, I review the current implementation by Micinski et al. (2018), and describe the more robust languages-as-libraries approach by Tobin-Hochstadt (2011).

RACETS uses macros to redefine core Racket forms to work with faceted values. The facets-tailored version of function application (for a single argument) is essentially implemented as

```scheme
(define-syntax (#%app stx)
  (syntax-case stx ()
    [(_ f a)
     #\'(if (facet? a)
           (fac
            (facet-labelname a))])
```
where the functions facet-labelname, facet-left, and facet-right are used to access the guard label, left branch, and right branch. This works on all instances of function application because #app, the name of the macro above, is a special form in Racket, and during macro expansion, all function applications are automatically wrapped with the #app form, so that (f a b) is re-written as (#app f a b). See Micinski et al. (2018) for the other facets-tailored core forms.

Implementing RACETS purely using macros suffers from a number of shortcomings: Racket’s built-in match macro and user-defined macros don’t work with faceted arguments, among other things. The reason why RACETS fails to support match and user-defined macros because RACETS does not know how to expand them properly (King, 2018). For instance, suppose we define our own macro called our-if-using-syntax-case that behaves just like Racket’s if:

```
(define-syntax (our-if-using-syntax-case stx)
  (syntax-case stx ()
    [(_ condition true-expr false-expr)
      #'(cond [condition true-expr]
              [else false-expr]))])
```

The macro takes three arguments: condition, true-expr, and false-expr.
If the condition evaluates to true, then true-expr is returned, otherwise false-expr is returned. Invoking

```
1 (our-if-using-syntax-case #t "true" "false")
```

returns "true" because the condition is #t. Say we define the faceted value `cond-facet`:

```
1 (define cond-facet (fac my-policy #t #f))
```

It would be logical for

```
1 (our-if-using-syntax-case cond-facet "true" "false")
```

to yield (fac my-policy "true" "false"). Unfortunately, this is not the case. During macro expansion, `our-if-using-syntax-case` expands to a regular Racket `if` statement:

```
1 ((if (deref cond-facet)
2   (let-values () 'true)
3   (let-values () 'false)))
```

This is wrong, as it should instead be distributing `if` over each of `cond-facet`'s facets. Unfortunately, macro expansion doesn’t know to do this, so invoking `our-if-using-syntax-case` with a faceted value simply yields the non-faceted value "true".

To support user-defined macros like `our-if-using-syntax-case` and `match` the RACETS system needs to comb through the entire program during macro expansion, identify core forms, and manually replace them with their appropriate facet-tailored versions defined by RACETS. These facet-tailored core forms take care of distributing forms across facets.
The key idea of the languages-as-libraries method (Tobin-Hochstadt, 2011) is to redefine the `#module-begin` form, which implicitly wraps all Racket modules during macro expansion, so that the entire source code of the module can be analyzed before it is evaluated. The source code can then be passed to a Racket function called `local-expand`, which invokes the macro expansion process to expand the source code, including user-defined macros, into a minimal subset of Racket called core Racket. To transform the source code, we only need to provide translations for the small set of core forms that constitute core Racket, rather than the much larger set of forms that may appear in regular, pre-expanded Racket. To demonstrate the expansion into core forms, the program `test-macro.rkt` below expands the source code and prints out the resulting core forms, before proceeding with evaluation.

```
1 #lang racket
2 (define-syntax (module-begin stx)
3   (syntax-case stx ()
4     [(_ forms ...)
5       (with-syntax ([(_ core-forms ...)
6       (local-expand
7         #'(#%plain-module-begin forms ...)
8         'module-begin
9         ()])
10         #'(#%plain-module-begin
11         (displayln '(core-forms ...))
12         core-forms ...))])
13   (provide (except-out (all-from-out racket))
```
#lang racket is declared at the top of the file because the program is written in Racket. However, this program is also a mini language (Tobin-Hochstadt, 2011), and as we will see later we can run an arbitrary Racket program in this language by just declaring #lang s-exp “test-macro.rkt” at the top of the file instead of #lang racket.

test-macro.rkt defines a macro called module-begin which rewrites its argument to be

```racket
#'(#%plain-module-begin
  (displayln '(core-forms ...))
  core-forms ...)))
```

where core-forms is defined by the with-syntax clause to be the result of invoking local-expand on the original syntax object. The macro then outputs a #%plain-module-begin form (rather than a #%module-begin form, to avoid infinite recursion during macro expansion) that wraps the original source syntax, after a call to displayln that prints at runtime the actual syntax object that was generated at compile time. Using the test-macro.rkt language, a program like

```racket
#lang s-exp "test-macro.rkt"
(define x 3)
(define y 2)
(+ x y)
```
would evaluate to (define-values (x) '3) (define-values (y) '2) (%app + x y)), which is what Racket refers to as a Fully Expanded Program.

0.4 Methodology

Here, I present a library called racets-macro.rkt that supports user-defined macros and match. The effect of the racets-macro.rkt module is to re-write the syntax tree so that for instance,

```
((if (deref cond-facet)
    (let-values () 'true)
    (let-values () 'false)))
```

becomes

```
(fac-if (deref cond-facet)
    (let-values () 'true)
    (let-values () 'false))
```

where fac-if is the facet-tailored version of the core form if defined in RACETS (Micinski et al., 2018).

First, we have fac-module-begin, which performs the whole-module expansion and analysis.

```
; expands program into core forms
(define-syntax (fac-module-begin stx)
  (syntax-case stx ()
    [(\_ forms ...) ([\_ forms ...]
      (let ([expanded
```
(local-expand '#( #%plain-module-begin forms ...) 'module-begin '(list #'define-values)))

For each form appearing in the source code, fac-module-begin expands it into its core forms using the Racket procedure local-expand, and then transforms those forms using a procedure called transform.

transform recursively walks the program’s syntax tree, detects core forms, and replaces them with their facet-tailored versions. Below is an abbreviated version of transform:

; swaps out Racket core forms with RACETS core forms
(define (transform stx)
  (syntax-case stx ()
    ([a b c d] ;if
      (check #'a #'if)
      (datum->syntax stx
        (cons #'fac-if
          (cons #'b
            (cons
              (transform #'c)
              (transform #'d))))))
    ([a b c] ;and
      (check #'a #'and)
      (datum->syntax stx
        (cons
          (transform #'c)
          (transform #'d)))))
(cons #'fac-and
    (cons #'b
        (cons
            (transform #'c)
            (transform #'d)))))

([a b c]; or
 (check #'a #'or)
 (datum->syntax stx
  (cons #'fac-or
    (cons #'b
        (cons
            (transform #'c)
            (transform #'d)))))

([a b]; quote
 (check #'head #'quote)
 #'(a b))

([a b ...]; other kinds of lists of forms
 (datum->syntax stx
  (cons #'a
    (map transform
      (syntax-e #'(b ...))))))

(dat #'dat)); single datum
As we can see, core forms if, and, and or are replaced by their respective facet-tailored versions: fac-if, fac-and, and fac-or. Cases for the core forms let-values, letrec-values, and plain-lambda are not shown, but they involve a more complex implementation. Each pattern has a guard clause that uses the helper function check defined on line 26 to check that the head of the matched form has the binding we are looking for. In Racket, users are allowed to alter the default meanings of reserved words, so it is necessary to match identifiers through their bindings, rather than through their names.

Since user-defined macros and match all expand to some variation of nested core forms, we can use the aforementioned forms and procedures to modify the expansions of most user-defined macros to support faceted arguments.

0.5 Results & Future Work

Integrating fac-module-begin, transform, and check into RACETS allows us to extend support for basic user-defined macros like our-if-using-syntax-case and instances of match such as the one shown below:

```
(define my-facet2 (ref (fac my-policy '(1 2 3) '(4 5 6))))
```
Here, my-facet2 is a faceted value with lists as branches. Applying \texttt{match} to my-facet2 yields \texttt{(fac my-policy 'sum-is-six 'sum-is-not-six)}, because the values on the positive branch add up to six but the values on the negative branch do not.

The extension of RACETS we present is not robust, as it still fails for certain complex macros that expand to multiple iterations of the core forms \texttt{let-values}, \texttt{letrec-values}, and other forms we did not account for. Addressing these issues would be the focus of future work.
Bibliography


Polikarpova, e. a. (2018). Enforcing information flow policies with type-targeted program synthesis. ACM.
