The Ergonomics of Faceted Execution

Ian Fisher

Advised by Kristopher Micinski

A thesis submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science in Computer Science

Haverford College
29 April 2018

Abstract

This thesis reviews several challenges in the use and implementation of faceted execution, a programming-language mechanism for enforcing privacy policies. I present a proof-of-concept of a module-rewriting technique for correctly handling mutable state in RACETS [8], an existing implementation of faceted execution in Racket, and I discuss how static typing and abstract interpretation could improve the safety and efficiency of faceted code.

Contents

1 Introduction

1.1 Faceted execution

Many software applications handle data that is protected by a privacy policy. On a social media website, for instance, a user's contact information may only be visible to the user's friends, while their name and profile picture may be public. Beyond social media, in industries like banking and healthcare the proper enforcement of privacy policies has serious legal and ethical implications. However, implementing privacy policies in code can be error-prone, as the privacy policies tend to become entangled with the application logic, and a significant proportion of the code ends up handling sensitive information.

Faceted execution is a programming-language mechanism that eases the implementation of privacy policies by decoupling the policies from the application's logic. Only a small percentage of the code in an application that uses faceted execution explicitly deals with privacy policies. The rest of the code can be written naively, while remaining secure. Faceted execution is thus an example of policy-agnostic programming [2].

The cornerstone of faceted execution is a data structure called the facet. 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. The actual value of the data to be protected is placed in v_H, and some default, non-private 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.

Researchers have pursued different strategies for implementing faceted execution. One strategy is to design a new programming language with faceted-execution primitives built-in. The Jeeves programming language is an example of this approach [13]. Another strategy is to integrate faceted execution into an existing language. The Racets programming language [8] adopts this strategy, by augmenting the Racket language with syntactic macros.

The rest of section 1 reviews the mechanics of faceted execution in Racets (though many of the concepts are more generally applicable). Section 2 presents a proof-of-concept of a module-rewriting technique that fixes erroneous handling of certain constructions involving mutable state. Sections 3 and 4 review how static typing and abstract interpretation, respectively, could improve the safety and efficiency of faceted code. Section 5 concludes the paper and offers suggestions for further research.

1.1.1 Core forms of faceted execution

Privacy policies are represented in Racets as predicates which take an argument and return a boolean value indicating whether the argument satisfies the policy or not. They are declared with the `let-label` form:

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

The code above creates a policy and binds it to the name `alice-policy`. Note the somewhat idiomatic usage of the `let-label` form: the general syntax is `(let-label name value body)`, which binds `name` to `value` and evaluates `body` with this new binding. In the declaration above, the body is simply the label name itself, so that the `let-label` form as a whole creates and then returns the label.

The policy enforces that only entities identifying themselves as "Alice" may view the high-confidentiality value of any facet protected by the policy.

A faceted data value is created with the `fac` form:

```
(define my-facet (fac alice-policy 42 0))
```

`my-facet` is defined with Alice's policy, the high-confidentiality value 42, 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.

```
(obs alice-policy "Alice" my-facet)
```

The expression above will evaluate to 42, as the argument `"Alice"` satisfies the facet's policy. By contrast, the expression below will evaluate to 0, since `"Bob"` does not satisfy the facet's policy.

```
(obs alice-policy "Bob" my-facet)
```

The policy passed to `obs` must match the policy that the facet was created with. In the case that the policies do not match, `obs` is a no-op. Each of the two calls to `obs` below, for instance, will return `my-facet` unchanged, since `my-facet` uses Alice's policy and not Bob's.

```
(obs bob-policy "Alice" my-facet)
(obs bob-policy "Bob" my-facet)
```

The `let-label`, `fac`, and `obs` forms expose the basic functionality of faceted execution in Racets. In a typical application, a set of policies is declared with `let-label`, sensitive data is enclosed with `fac`, and the points at which sensitive data must be revealed do so using `obs`. The rest of the code does not need to explicitly deal with privacy policies.

1.1.2 Passing facets to functions

A major advantage of faceted execution is that policy-naive code can be applied to faceted values. Suppose that a square function were defined as follows:

```
(define (square x) (* x x))
```

`square` pays no special regard to privacy considerations—it is written exactly as it would be in an application with no privacy policies at all. Under normal circumstances, `(square (fac p 4 0))` would fail, because `square` expects a numeric argument, not a facet. However, in RACETS, this function call is invisibly transformed to become `(fac p (square 4) (square 0))`, distributing the function application across the two values of the facet, and preserving the data's privacy policy in the return value.

This transformation, which applies to all function applications in RACETS, ensures that functions like `square` can be applied to faceted values even when written for non-faceted values.

1.1.3 Nested facets

Faceted values may be nested for more finely-grained control over the available views of data. While a single facet encloses only two views of the data, a nested facet may enclose an arbitrary number of views. For example, Alice could define a nested facet that specifies her location in three degrees of granularity:

```
(define location-facet
  (fac alice-policy
    "370 Lancaster Ave, Haverford PA"
    (fac bob-policy
      "Haverford, PA"
      "Pennsylvania")))
```

Alice is able to view the full street address of her location. Bob (or anyone else satisfying Bob's policy) may see her town, and anyone else may only see her state of residence. The structure of the nested facet may be visualized as a tree, where the left branches are the high-confidentiality values, the right branches are the low-confidentiality values, the leaves are the non-faceted values, and the internal nodes are the policies. Figure 1 represents the facet defined above.

Alice observes her nested facet in the usual way:

```
(obs alice-policy "Alice" location-facet)
```

Bob must make two calls to `obs` to fully resolve the facet's value:

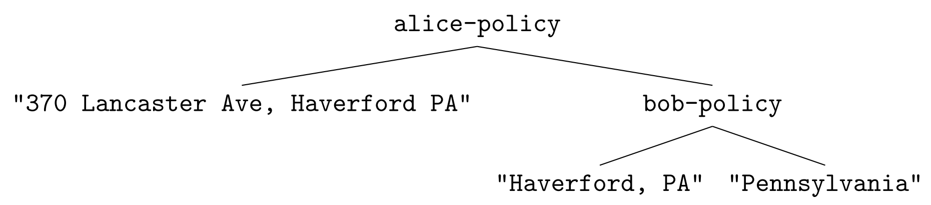

Figure 1: A nested facet

```
(obs alice-policy
    "Bob"
    (obs bob-policy "Bob" location-facet))
```

As before, Bob must ensure that the policy he passes to each `obs` call matches the policy of the facet. In this case, the outer facet uses Alice's policy and the inner facet uses Bob's policy, so the calls to `obs` must be organized likewise.

1.1.4 Faceted structs

Compound data structures such as structs may be faceted. However, the behavior of faceted structs in Racets is somewhat counterintuitive. Take the following example:

```
(struct employee (name position salary))
(define bob
  (employee "Bob" "manager" (fac bob-policy 70000 0)))
```

The call to the `employee` constructor is handled by Racets the same way that any function call is: by branching execution on both values on the facet. Consequently, the return value is `(fac bob-policy (employee "Bob" "manager" 70000) (employee "Bob" "manager" 0))`, i.e. a facet that wraps two different instantiations of the `employee` struct, rather than a structure containing a faceted value for the `salary` field, as one might expect. Bob's faceted salary can be accessed like so:

```
(employee-salary (obs bob-policy "Bob" bob))
; or, equivalently:
(obs bob-policy "Bob" (employee-salary bob))
```

A surprising consequence is that `obs` is required to observe the name field (and any other field) of the `employee` object, even though it was not explicitly faceted in the constructor.

```
(obs bob-policy "Bob" (employee name bob))
```

Racets programs must take care to observe this subtlety when working with faceted structs.

1.1.5 Faceted lists

Using faceted values with lists involves similar complications. Consider the following example:

```
(define grades (list))
(set! grades (cons (fac alice-policy 84 0) grades))
```

The programmer likely intended for `grades` to be of the form `(list (fac alice-policy 84 0))`, i.e. a regular list containing a single facet. However, the real value of `grades` after the `set!` operation is `(fac alice-policy (list 84) (list 0))`, because the call to `cons` is transformed into

```
(fac alice-policy (cons 84 grades) (cons 0 grades))
```

per the standard function application transformation. Imagine another grade was added to the list, like so:

```
(set! grades (cons (fac bob-policy 73 0) grades))
```

Then the value of the list would be

```
(fac bob-policy
  (fac alice-policy (list 73 84) (list 73 0))
  (fac alice-policy (list 0 84) (list 0 0)))
```

encompassing four different possibilities: satisfying both Alice and Bob's policies, satisfying only Alice's policy, satisfying only Bob's policy, or satisfying neither. The tree in figure 2 illustrates this structure diagrammatically.

Programmers may find it useful to convert a faceted list into a list of non-faceted values. `obs-list` accomplishes this:

```
(define (obs-list faceted-list policy-list arg)
  (if (empty? policy-list)
    faceted-list
    (obs
      (car policy-list)
```

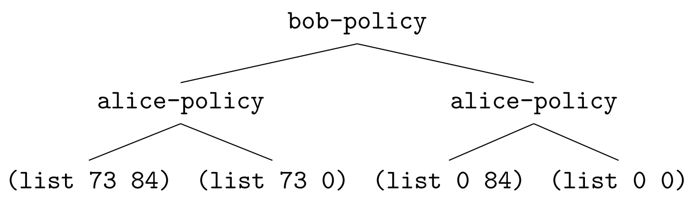

Figure 2: A nested facet containing lists

```
      arg
      (obs-list
        faceted-list
        (cdr policy-list)
        arg))))])
```

`obs-list` takes in a faceted list, a list of policies which must correspond index-by-index to the policies attached to the elements of the faceted list (i.e., so that the i'th policy corresponds to the i'th facet in the list),[1] and an argument to pass to the policy predicates. It traverses the policy tree tail-recursively, observing a single facet at each step.

This section and the previous one have illustrated usability issues with the interaction of faceted execution and data structures, stemming from the generic treatment of faceted function application. For the sake of programming ergonomics, future implementations of faceted execution may wish to not treat function application quite so generically, so as to allow structure-creating forms like `list` and struct constructors to behave more intuitively with faceted arguments.

1.2 Syntactic macros

Since syntactic macros figure heavily in the implementation of Racets, a brief introduction to the Racket macro system is given here. Readers are referred to the “Fear of Macros” tutorial [6] for a much more comprehensive treatment.

A syntactic macro is a syntax-transforming procedure that runs at compile-time. Syntactic macros allow programmers to define new syntactic structures without having to modify the language's parser.

At compile time, the Racket compiler searches through the source code for invocations of user-defined macros, evaluates the macros where they appear, and substitutes the result into the original source code. It then continues to recursively search the syntax tree, expanding macros until no more are left. This process is known as *macro expansion*.

[1]Alternatively, if all elements of the list use the same policy, `obs-list` could be modified to take a single policy rather than a list of policies.

The definition of a macro resembles a function that receives a syntax object as input and returns another syntax object as output, except that `define-syntax` is used instead of `define`. The following macro uses the `syntax` constructor for syntax objects to return a new syntactic form, ignoring its input:

```
(define-syntax (say-hello stx)
  (syntax (displayln "hello")))
```

A macro is invoked just like a regular function:

```
; Expands to (displayln "hello") -- the input to the
; macro is ignored.
(say-hello (+ 2 2))
```

Unlike a regular function, a macro's argument is a syntax tree rather than a concrete value. In the call to `say-hello` above, the `stx` parameter would be bound to the entire syntax object of `(say-hello (+ 2 2))`, and *not* the concrete value 4.

Syntax objects can be converted to and from lists for easier manipulation, using `syntax->datum` and `datum->syntax`. This functionality allows us to implement a macro that reverses its arguments:

```
1 (define-syntax (reverse-syntax stx)
2   (datum->syntax stx
3                  (reverse (cdr (syntax->datum stx)))))
4
5 ; Expands to (+ 20 22) at compile time, which evaluates
6 ; to 42 at runtime
7 (reverse-syntax 20 22 +)
```

In the macro invocation on line 6, `stx` will be bound to the entire syntax tree of the macro call, so that `(syntax->datum stx)` returns `'(reverse-syntax 20 22 +)` and not `'(20 22 +)`. Consequently, we must call `cdr` on this list before reversing it, to remove the name of the macro.

Since syntactic 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. Lazy evaluation of this sort could not be implemented by a regular function, because a regular function has no control over the evaluation of its arguments—they are always evaluated before they are passed.

```
(define-syntax (my-and stx)
  (syntax-case stx ()
    [(_ a b)
     #'(if a b #f)]))
```

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,[2] and the symbols a and b match arguments to the macro.

Macros that use syntax-case can often be abbreviated with define-syntax-rule. Using define-syntax-rule, my-and could be re-written as

```
(define-syntax-rule (my-and-2 a b)
  (if a b #f))
```

Macros defined with define-syntax-rule can have multiple parameters, which bind to the syntactic arguments to the macro. The body of the macro is implicitly converted from an S-expression into a syntax object.

One final point about syntactic macros in Racket is that they are hygienic, meaning that identifiers introduced in the macro are protected from conflicting with identical symbols in the original source code. Consider the following macro, which takes two variables and swaps their values:

```
(define-syntax-rule (swap a b)
  (let ([tmp a])
    (set! a b)
    (set! b tmp)))
```

If swap were called with a variable tmp as its argument, e.g.

```
(define tmp 42)
(define y 0)
(swap 42 0)
```

then one might expect that, by simple textual substitution, it would expand to

```
(define tmp 42)
(define y 0)
```

[2]The use of an underscore is purely conventional, to indicate that the macro does not care about the value it matches. syntax-case itself treats underscores the same as any other identifier in a syntax pattern.

```

; Wrong!
(let ([tmp tmp])
  (set! tmp y)
  (set! y tmp))
```

so that the first `set!` operation would affect the symbol `tmp` defined in the macro, rather than the `tmp` defined by the user, and consequently fail to swap the variable's values. In actual fact, the macro system rewrites the identifier `tmp` in the macro to a name which it can guarantee will not collide with any existing identifier in the program, as shown below.

```
; Correct -- tmp in the macro is rewritten as tmp:22
(let ([tmp:22 tmp])
  (set! tmp y)
  (set! y tmp:22))
```

At runtime the expanded macro thus correctly swaps the variable's values, thanks to the guaranteed hygiene of Racket macros.

2 Extending Racket with faceted execution

Having seen the interface that Racets exposes to its users, and with the requisite background in syntactic macros, we now turn to the implementation of Racets. In this section, I review the current implementation, and present a more flexible and powerful approach using Racket's `#lang` mechanism, as well as a concrete proof-of-concept illustrating a crucial module-rewriting technique that resolves an issue with the semantics of mutable state.

2.1 Racets with macros

Racets uses macros to redefine core Racket forms to make them sensitive to faceted values. For example, the transformation of function application (for a single argument) could be implemented by[3]

```
(define-syntax (#%app stx)
  (syntax-case stx ()
    [(_ f a)
     #'(if (facet? a)
           (fac
```

[3] The actual implementation in [8] is more complex, because it handles multiple arguments, as well as the possibility that the function itself is faceted.

```
          (facet-labelname a)
          (f (facet-left a))
          (f (facet-right a)))
        (f a))]))
```

using the private functions `facet-labelname`, `facet-left`, and `facet-right` to access the internal fields of the facet.

`#%app`, the name of the macro above, is a special form in Racket. During macro expansion, all function applications are transformed to use the `#%app` form, so that `(f a b)` is re-written as `(#%app f a b)`. This transformation occurs in plain Racket, independently of RACETS. Defining a macro for `#%app` allows all function applications to be captured and transformed, which is exactly what the semantics of RACETS require.

Other macros are provided for a subset of the core forms of Racket. See [8] for details.

2.1.1 Shortcomings

Implementing RACETS using macros suffers from two shortcomings: certain constructions involving `set!` cannot be handled correctly, and the RACETS macros may interfere with user-defined macros.

To correctly handle `set!` forms with faceted values, the RACETS system needs to wrap (some) identifiers with extra conditional logic. To see why, consider the following code snippet:

```
1 (define x 0)
2 
3 (if fct
4   (set! x 100)
5   (set! x 50))
6 
7 (square-root x)
```

Suppose that `fct` is a facet belonging to Alice whose high-confidentiality value is true and whose low-confidentiality value is false. Upon execution of the `if` statement, the value of `x` will be a box containing `(fac alice-policy 100 50)`. However, `square-root` expects an integer argument, and not a box, so the call to it on line 7 will fail. Note that the problem is not that `x` is invisibly transformed into a facet, which happens frequently in policy-agnostic faceted code and is handled by the mechanisms of faceted execution, but that it becomes a box containing a facet.

A simple, brute-force method of handling cases like these is to wrap all bare identifiers in conditional logic that checks if the identifier's value is a box containing a facet, and unboxes it if it is.[4]

One could imagine that there is a `#%identifier` form in Racket which wraps all identifiers at some point in the macro expansion process, analogous to the `#%app` form for function applications. In that case, Racets could include a macro for `#%identifier` that implements the necessary logic. Unfortunately, there is no such form: identifiers are “raw” in the syntax and cannot be targeted individually and exclusively by a syntactic macro. A different approach must be used.

The second issue is that Racets may interfere with other macros, for instance if a programmer defined a macro for the `#%app` form themselves. As it turns out, the same approach—the languages-as-libraries approach—that solves the previous issue addresses this problem as well.

2.2 Racets as a language-library

All plain Racket files begin with a `#lang racket` declaration. The `#lang` statement may also be used to declare the use of other languages, e.g. `#lang typed/racket` for Typed Racket [11], and it may be used to delegate parsing of the file to another Racket file. In the latter case, it is written as `#lang s-exp "m.rkt"`, where `m.rkt` is the name of the Racket file to which parsing has been delegated. `m.rkt` defines syntactic macros which are used in the expansion of the file that includes the `#lang s-exp "m.rkt"` declaration.

The languages-as-libraries approach [10] uses `#lang` to implement new domain-specific languages which are compatible with plain Racket code, and which can reuse the functionality exposed by the Racket compiler, greatly simplifying their implementation. The key idea of languages-as-libraries is to redefine the `#%module-begin` form, which implicitly or explicitly wraps all Racket modules, so that the entire source code of the module can be intercepted 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 into a minimal subset of Racket called Fully-Expanded Racket [1], expanding all user-defined macros along the way.

The language-library then only needs to provide translations for the small set of core forms that constitute Fully-Expanded Racket, rather than the much larger set of forms that may appear in regular Racket. Furthermore, it does not need to worry about conflicts with user-defined macros, since they have already been expanded by `local-expand`.

2.2.1 Small example of a language-library

The following minimal but complete example of the languages-as-libraries idea prints out the fully-expanded abstract syntax tree of a program before running it normally:

[4]Section 4 will explore a technique for mitigating the resulting performance hit.

```
#lang racket

; Fully expands the module and prints out its abstract
; syntax tree, before running it normally.
(define-syntax (module-begin stx)
  (syntax-case stx ()
    [(_ forms ...)
    ; Bind core-forms to the result of macro-expanding
    ; forms.
    (with-syntax ([(_ core-forms ...)
                   (local-expand
                     #'(#%plain-module-begin forms ...)
                     'module-begin
                     '())])
      #'(#%plain-module-begin
          (displayln '(core-forms ...))
          core-forms ...))]))

; Export everything from the regular Racket language,
; except replace #%module-begin with our own
; implementation.
(provide (except-out (all-from-out racket)
                     #%module-begin)
         (rename-out [module-begin #%module-begin]))
```

If this program were saved in a file called `print-ast.rkt`, then other files could be written in the language by declaring `#lang s-exp "print-ast.rkt"` at the top of the file.

The file `print-ast.rkt` begins with the standard `#lang racket` declaration, since it itself is written in Racket. It then defines a macro called `module-begin` which rewrites its argument to be

```
#'(#%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.

2.2.2 Proof-of-concept of Racets as a language-library

As a proof-of-concept of implementing Racets with the language-as-a-library approach, this section presents a library called `wrap-ident.rkt` that rewrites identifiers in the source code in roughly the manner that the real RACETS implementation requires.

The effect of the `wrap-ident.rkt` module is to re-write the syntax tree so that

```
#lang s-exp "wrap-ident.rkt"

(define x 10)
(define y 32)
(displayln (+ x y))
```

becomes

```
(define x 10)
(define y 32)
(displayln ((var-wrapper +) (var-wrapper x)
                            (var-wrapper y)))
```

where `var-wrapper` is defined with the logic outlined in section 2.1.1:

```
(define (var-wrapper v)
  (if (and (box? v) (facet? (unbox v)))
    (unbox v)
    v))
```

The core of `wrap-ident.rkt` is a procedure called `transform-syntax`, which recursively walks the program's syntax tree, detects bare identifiers, and wraps them with `var-wrapper`. Most forms in Fully-Expanded Racket can be transformed simply by recursively transforming each argument. Some forms need to be treated specially. An example is `set!`: `(set!  x 10)` could not be re-written as `(set!  (var-wrapper x) 10)`, as the first argument to `set!` must be a bare identifier. `transform-syntax` therefore contains patterns to match special-case forms, followed by a default case for any other list of forms, followed by a case for any individual form, which performs the actual work of identifying and re-writing identifiers. An abbreviated implementation is given here:

```
(define (transform-syntax stx)
  (syntax-case stx ()
    ; set!
    ([head id expr]
     (check-ident #'head #'set!)
     #`(head id #,(transform-syntax #'expr)))

    ; provide
    ([head a ...]
     (check-ident #'head #'#%provide)
     stx)

    ; #%plain-lambda
    ([head formals expr ...]
     (check-ident #'head #'#%plain-lambda)
     (datum->syntax stx
                    (cons #'head
                          (cons #'formals
                                (map transform-syntax
                                     (syntax-e #'(expr ...)))))))

    ; quote
    ([head datum]
     (check-ident #'head #'quote)
     #'(head datum))

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

    ; Any other individual form.
    (default
      (if (identifier? #'default)
        #`(var-wrapper #,#'default)
        #'default))))
```

The code listing above is not exhaustive: special cases for `let-values` and letrec-values, among others, are omitted for the sake of brevity as their syntax requires complex pattern matching.

Each pattern has a guard clause that uses the helper function check-ident to check that the head of the matched form is bound to the same variable as the form to be re-written. check-ident is defined below. It uses free-identifier=? to check its argument's binding. It is important to use this function rather than comparing the textual content of the identifiers, because Racket does not guarantee the textual content will be the same. For instance, an identifier may have the surface form of lambda but actually be bound to #%plain-lambda. free-identifier=?, unlike a comparison of symbolic equality, is sensitive to this distinction.

```
(define (check-ident ident expected)
  (and (identifier? ident)
       (free-identifier=? ident expected)))
```

The final piece is the whole-module rewrite rule:

```
(define-syntax (module-begin stx)
  ; Intercept the entire module and transform it using
  ; `transform-syntax` on the fully-expanded syntax tree
  ; (which ensures that user macros are expanded first).
  (syntax-case stx ()
    [(_ forms ...)
     (let ([expanded
            (local-expand #'(#%plain-module-begin forms ...)
                          'module-begin
                          '())])
       (transform-syntax expanded))]))

(provide (except-out (all-from-out racket)
                     #%module-begin)
         (rename-out [module-begin #%module-begin]))
```

The module-begin macro invokes local-expand on the module contents, and then calls transform-syntax to transform the resulting fully-expanded syntax tree.

If this proof-of-concept were extended to handle the remaining core forms of Fully-Expanded Racket, and augmented with the original set of Racets macros, it would yield a more robust implementation of faceted execution in Racket.

3 Faceted execution and type theory

Since RACETS, like Racket, is dynamically typed, errors in the use of faceted values (such as supplying the wrong policy to `obs`) are not caught until the invalid operation is attempted at runtime. A static type system for faceted values could detect many programmer errors at compile time. This section presents an introduction to formal type theory and indicates how it might be applied to faceted execution.

A type system is "a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute" [9]. The program behaviors that a type system are meant to prevent are generally what are called type errors: attempted operations on values for which the operation is not defined.

Naturally, type systems cannot catch all errors: some undesirable program behaviors cannot be detected in the general case (e.g., looping infinitely is not detectable due to the uncomputability of the halting problem), and type systems will sometimes reject constructs which are actually safe. Nonetheless, a well-designed type system is an effective tool for catching programmer errors.

Formal type theory will be presented through the example of the typed lambda calculus. The following exposition is based on chapter 9 of [9].

The lambda calculus is a simple mathematical model of computation in which the only data type is the function and the only operation is function application. Despite its simplicity, any possible computation can be expressed in the lambda calculus, per the Church-Turing thesis. The syntax of the lambda calculus comprises three kinds of terms: variables (x, y, etc.), anonymous functions, also known as abstractions ($\lambda x.x$), and function applications ($t\ t$). The notation $\lambda x.x$ is equivalent to the more traditional notation of $f(x) = x$, with the advantage of not requiring the function to be named. Function application of the form $t_1\ t_2$ is equivalent notationally to $t_1(t_2)$, i.e. the application of the function t_1 to the argument t_2. The body of a lambda function may contain any of the three syntactic forms of the language, so for example $\lambda x.\lambda y.(\lambda z.z\ x)$ is a valid term in the lambda calculus whose outermost abstraction contains another abstraction, which in turn contains an application ($\lambda z.z$) to a variable (x). For clarity, the application is wrapped in parentheses.

The typed lambda calculus has the same syntax as the untyped lambda calculus, except that lambda abstractions carry a type annotation for their parameter, written as $\lambda x : T.x$, where T is a type. A *type annotation* is an annotation supplied by the programmer that indicates the intended type of a variable or expression. Not all typed languages require type annotations, but requiring them simplifies the specification of the type system.

A type system for the lambda calculus (and indeed for any programming language) must assign a type to each syntactic construction in the language. These assignments are formally known as type rules. Type rules are written in the form

$$\frac{\text{hypotheses}}{\text{conclusion}}$$

where *hypotheses* is a set of assumptions and *conclusion* is a type judgment that follows from the hypotheses.

The rule for the types of variables in the lambda calculus is[5]

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

Γ stands for an assignment function that maps from a finite number of variable names to their values. The notation $\Gamma \vdash t : T$ expresses the three-place relation that syntactic term t has type T given the assignment Γ. The type rule for variables simply states that if a variable is paired with a certain type T in the assignment function, then T is the variable's type.

The rule for lambda abstraction is a bit more complicated:

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2}$$

In the hypothesis, the expression $\Gamma, x : T_1$ should be read as "Γ augmented with the assignment of type T_1 to x." It is assumed that the variable x is not already in Γ; if it is, it can be renamed. The full hypothesis of the abstraction rule states that the body of the lambda function has type T_2 with the given assignment function. The conclusion of the abstraction rule states that the lambda function as a whole has the complex type $T_1 \to T_2$, the type of a function from values of type T_1 to values of type T_2.

Finally, the last syntactic form of the language, application, has the type rule

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 \; t_2 : T_{12}}$$

Given that the function has type $T_{11} \to T_{12}$ in Γ, and the argument has type T_{11}, then the application as a whole has type T_{12}.

The three type rules presented above can be applied to any expression in the typed lambda calculus to detect whether it is well-typed, and if it is, to determine its concrete type. Thus the type system is able to statically detect type errors in the lambda calculus.

Type systems for real programming languages are significantly more complex, but involve the same basic formalism of inference rules and type judgments for each syntactic form in the language. As such, this formalism could be deployed to augment RACETS with a static type system that detects programming errors specific to faceted execution.

[5] The type rules are taken from Figure 9-1 on p. 103 of [9].

4 Faceted execution and abstract interpretation

Another practical problem with faceted execution is that faceted code can run significantly more more slowly than non-faceted code. Functions applied to facets must be evaluated twice, once with the high-confidentiality value and once with the low-confidentiality value, resulting in a performance hit for faceted code.[6] This performance hit can be mitigated by static analysis. A static analyzer might, for instance, be able to prove that a facet is only ever resolved to its high-confidentiality value, in which case the high-confidentiality value could be substituted for the facet as a whole, skipping double evaluation wherever it is passed to a function.

One technique for static analysis is abstract interpretation [7], wherein the computation of a program is modeled with abstract categories instead of concrete values (e.g., "negative number" instead of -10) so that semantic properties of the program can be deduced by a decidable procedure [3].

4.1 Deriving abstract interpreters from abstract machines

Abstract interpreters can be derived from abstract machines using the "abstracting abstract machines" technique presented in [12]. An *abstract machine* is a formal mathematical object that models a computing machine. Abstract machines consist of a set of elements that comprise its state, and transition rules that determine how the machine moves from state to state.

Three abstract machines of increasing resemblance to real-world technology will be presented here: the CEK, CESK, and CESK* machines. Though the language that these machines evaluate (the untyped lambda calculus) is superficially very different from real programming languages, in principle the mechanisms of function definition, application, and lexical closure are the core of many actual languages, and at any rate the lambda calculus is much closer to a realistic programming language than, e.g., the finite transition systems that underlie the computation of a Turing machine are.

The CEK machine has three elements in its state: c, an expression in the lambda calculus; ρ, an environment that maps from symbols to values; and κ, a continuation (the letters of the acronym "CEK" stand for the elements of the state). All values in the lambda calculus are closures, which are represented by pairs of lambda functions and environments.

A continuation is a mathematical structure for representing control flow. Intuitively, a continuation encapsulates what the machine should do next after evaluating the current expression. For the lambda calculus, the sequence of machine operations is very simple, as only function application (of the three syntactic terms) requires any control flow at all: the function is evaluated first, followed by the argument, and then the body of the function. At each step, the continuation stores what the next step is.

[6]It also complicates the use of functions with side-effects, which is an open problem in Racets.

In the CEK machine, continuations come in three forms: $\mathbf{ar}(e, \rho, \kappa)$ for evaluating an argument, $\mathbf{fn}((\lambda x.e), \rho, \kappa)$ for evaluating the body of a function, and **mt** ("empty") for halting computation. In the first two forms, ρ is the environment in which the expression is to be evaluated, and κ is the continuation that is to be followed afterwards.

The entirety of the operation of the CEK machine is described by four transition rules that map from one machine state to another.[7] The first rule is for evaluating variables:

$$1.\ \langle x, \rho, \kappa\rangle \rightarrow \langle v, \rho', \kappa\rangle,\ \text{given that } x \text{ is bound to } (v, \rho') \text{ in } \rho$$

The new expression to evaluate is the closure that the variable maps to in the environment. The old environment is replaced with the closure's environment. The original continuation is preserved, since evaluating variables does not involve a change in control flow.

The second rule is for evaluating function applications:

$$2.\ \langle (e_0\ e_1), \rho, \kappa\rangle \rightarrow \langle e_0, \rho, \mathbf{ar}(e_1, \rho, \kappa)\rangle$$

The first expression to be evaluated in a function application is the function itself, so this function (e_0 in the formula) becomes the expression in the output state. The environment in which the function is evaluated is the same in which the function application as a whole is evaluated, so ρ is preserved. The new continuation directs the abstract machine to evaluate the argument next, in the same environment, followed by the original continuation κ.

The last two rules handle the evaluation of values, i.e. closures. If the current continuation is of the **ar** form, the following rule applies:

$$3.\ \langle v, \rho, \mathbf{ar}(e, \rho', \kappa)\rangle \rightarrow \langle e, \rho', \mathbf{fn}(v, \rho, \kappa)\rangle$$

The old expression and environment are replaced with the expression and environment provided by the continuation. After evaluating the argument, we want to evaluate the body of the function, so we provide a **fn** continuation that encapsulates the function itself, the original environment, and the original continuation.

The last rule applies to evaluating values with an **fn** continuation:

$$4.\ \langle v, \rho, \mathbf{fn}((\lambda x.e), \rho', \kappa)\rangle \rightarrow \langle e, \rho'[x \mapsto (v, \rho)], \kappa\rangle$$

The body of the function, e, is the next expression to be evaluated, so it goes in the expression slot of the output state. The environment is the one indicated by the old continuation, augmented with a binding of the function's parameter to the value of its argument, which must be in the old expression slot per the previous rule. We preserve the old continuation.

[7] These rules are reproduced and slightly modified from Figure 1 on p. 2 of [12]. See [5] for a different formulation of the CEK machine's transition rules.

In addition to state-mapping rules, an abstract machine needs an *injection function* that provides its initial state, given an expression to evaluate. For the CEK machine, the injection function is defined as

$$inj_{CEK}(e) = \langle e, \emptyset, \mathbf{mt} \rangle$$

The environment is initially empty, and the continuation is **mt**, because the next thing to do after evaluating the expression is to halt.

The transition rules may be clarified by walking through their application in a concrete example. Suppose we want to evaluate the lambda expression $(\lambda x.x\ \lambda y.y)$ (which should evaluate to $\lambda y.y$). We begin by calling inj_{CEK} to get the initial state of the machine:

$$s_0 = \langle (\lambda x.x\ \lambda y.y), \emptyset, \mathbf{mt} \rangle$$

s_0 matches rule 2, so we apply the rule to yield the next state. As expected, we are evaluating the function $(\lambda x.x)$ first, and the continuation indicates that we must evaluate the argument afterwards:

$$s_1 = \langle \lambda x.x, \emptyset, \mathbf{ar}(\lambda y.y, \emptyset, \mathbf{mt}) \rangle$$

s_1 in turn matches rule 3, which we apply below. We are now evaluating the argument, and the continuation tells us that evaluating the body of the function is next.

$$s_2 = \langle \lambda y.y, \emptyset, \mathbf{fn}(\lambda x.x, \emptyset, \mathbf{mt}) \rangle$$

s_2 matches rule 4. We bind the value of the argument to the function's parameter in the environment, and evaluate the body in this new environment. Since evaluating the body is the last step in evaluating the function application as a whole, the continuation points to halting computation as expected.

$$s_3 = \langle x, \{x : (\lambda y.y, \emptyset)\}, \mathbf{mt} \rangle$$

s_3 matches rule 1:

$$s_4 = \langle \lambda y.y, \emptyset, \mathbf{mt} \rangle$$

We have now reached a state whose expression is a value and whose continuation is **mt**, so computation halts. As expected, the value $\lambda y.y$ is in the expression slot of the final slot.

The CEK machine can be augmented with a store to represent mutable state. A store maps addresses to values, and can be imagined as the abstract equivalent of a computer's memory unit.

The CEK machine augmented with a store is known as the CESK machine [4]. [12] present a slight modification to the CESK machine, in which continuations are allocated in the store in order to make the machine more amenable to conversion into an abstract interpreter. This machine is known as the CESK* machine.

The state of the CESK* machine consists of c, ρ, σ, a, where c is defined as for the CEK machine, ρ is modified to map from symbols to addresses instead of values, the store σ maps from addresses to values, and a is the address of a continuation in the store.[8]

The injection function for the CESK machine is slightly (but only slightly) more complex than inj_{CESK}:

$$inj_{CESK*}(e) = \langle e, \emptyset, \{a_0 : \mathbf{mt}\}, a_0\rangle$$

The environment is initially empty. The store contains the **mt** continuation, and the continuation address slot points at this continuation.

The four rules for the behavior of the CESK* machine are likewise similar.[9] Several of the rules use $alloc(\sigma)$, a function which returns a new address in the store that has not been used yet.

The first rule, for evaluating variables, is almost identical:

$$1.\ \langle x, \rho, \sigma, a\rangle \to \langle v, \rho', \sigma, a\rangle,\ \text{given that } (v, \rho') = \sigma(\rho(x))$$

The second rule, for function application, differs only in that the continuation is allocated in the store:

$$2.\ \langle (e_0\ e_1), \rho, \sigma, a\rangle \to \langle e_0, \rho, \sigma[b \mapsto \mathbf{ar}(e_1, \rho, a)], b\rangle \text{ where } b = alloc(\sigma)$$

The third and fourth rules again differ only in the use of the store:

$$3.\ \langle v, \rho, \sigma, a\rangle \to \langle e, \rho', \sigma[b \mapsto \mathbf{fn}(v, \rho, c)], b\rangle \text{ where } b = alloc(\sigma)$$
$$\text{given that } a \text{ maps to } \mathbf{ar}(e, \rho', c) \text{ in } \sigma$$

$$4.\ \langle v, \rho, \sigma, a\rangle \to \langle e, \rho'[x \mapsto b], \sigma[b \mapsto (v, \rho)], c\rangle$$
$$\text{given that } a \text{ maps to } \mathbf{fn}((\lambda x.e), \rho', c) \text{ in } \sigma$$

The CESK* machine is thus quite similar in principle to the CEK machine, but the use of a store for values and continuations allows the construction of an abstract interpretation from the CESK* machine to proceed more smoothly.[10] The reason is that abstract interpretation must terminate to be useful in practice. The abstract interpreter can enforce that the store is finite, so moving values and continuations to the store ensures that the interpretation will terminate, which otherwise could not be guaranteed if environments and continuations were arbitrary recursive structures unbound to any finite container.

The abstract CESK* interpreter is defined similarly to the CESK* machine, except that the store maps to sets of values instead of concrete values, and

[8]For the regular CESK machine, a is instead κ with the same meaning as for the CEK machine.

[9]These rules are reproduced and slightly modified from Figure 3 on p. 4 of [12].

[10]In [12], the authors tweak the CESK* machine to use time-stamps before they derive the abstract interpreter. For simplicity of exposition, this modification has been omitted.

the machine transitions between sets of states instead of individual states. The interpreter continues transitioning until it reaches a fixed-point, when applying its transition function to each individual states yields the same set of states as before. Once the abstract CESK* interpreter has terminated, its store can be inspected to determine semantic properties of the program.

Formally, the abstract CESK* interpreter is a 4-tuple $(c, \rho, \hat{\sigma}, a)$, where c, ρ and a are defined the same as for the CESK* machine, and $\hat{\sigma}$ maps from addresses to sets of values.

The behavior of the abstract CESK* interpreter is defined by four transition rules. Some of these rules make use of an allocation function $\widehat{alloc} : \hat{\Sigma} \times Kont \to Addr$, which takes in an abstract interpreter state and a continuation in order to decide what address to return. Its implementation is left unspecified here.

The first transition rule, for variables, is essentially identical to the analogous rule for the CESK* machine:[11]

$$1.\ \langle x, \rho, \hat{\sigma}, a\rangle \to \langle v, \rho', \hat{\sigma}, a\rangle,\ \text{given that } (v, \rho') = \sigma(\rho(x))$$

The remaining three rules are where the important difference between the CESK* machine and the abstract CESK* interpreter comes to light. Rather than assigning directly to the store, the new mapping is unioned with the existing store, so that the store in the output state contains both the new mapping and any old mapping that the new mapping might have otherwise overwritten.

$$2.\ \langle (e_0\ e_1), \rho, \hat{\sigma}, a\rangle \to \langle e_0, \rho, \hat{\sigma} \cup [b \mapsto \mathbf{ar}(e_1, \rho, a)], b\rangle \text{ where } b = \widehat{alloc}(\varsigma, \kappa)$$

$$3.\ \langle v, \rho, \hat{\sigma}, a\rangle \to \langle e, \rho', \hat{\sigma} \cup [b \mapsto \mathbf{fn}(v, \rho, c)], b\rangle \text{ where } b = \widehat{alloc}(\varsigma, \kappa)$$
$$\text{given that } \mathbf{ar}(e, \rho', c) \in \hat{\sigma}(a)$$

$$4.\ \langle v, \rho, \hat{\sigma}, a\rangle \to \langle e, \rho'[x \mapsto b], \hat{\sigma} \cup [b \mapsto (v, \rho)], c\rangle$$
$$\text{given that } \mathbf{fn}((\lambda x.e), \rho', c) \in \hat{\sigma}(a)$$

Other than modifying the store in the manner described above, the transition rules for the abstract CESK* interpreter are more or less the same as for the CESK* machine.

When the abstract interpreter has terminated, its store can be inspected to reason about the semantics of the program. For example, if every element of the set that address a maps to is a negative number, then we can conclude that the variable or variables corresponding to a will always be negative.

We are now in a position to understand why store-allocated continuations are crucial for converting the CESK* machine into an abstract interpreter. Abstract interpretation must terminate to be useful in practice. One way of enforcing termination is by keeping the store finite, which can be accomplished by defining the allocation function $\widehat{alloc}$ in such a way that its co-domain is finite. If

[11]These rules are reproduced from Figure 5 on p. 5 of [12] and modified to apply to the CESK* interpreter instead of the time-stamped CESK* interpreter originally presented in the paper.

continuations were outside of the store, they would be recursive structures that could potentially nest infinitely. Once moved to the store, their finite cardinality is guaranteed by the broader finite cardinality of the store itself.

In summary, computation of the lambda calculus can be described mathematically with the CEK, CESK, and CESK* machines. The CESK* machine can be converted into an abstract interpreter which deduces semantic properties of programs in the lambda calculus, while guaranteeing termination.

4.2 Abstract interpretation of Racets

Recall that the original motivation for abstract interpretation was to improve the runtime performance of programs by statically proving facts about the program's semantics that could be used for compile-time optimizations. The extended Racets implementation presented in section 2 has a particularly high runtime overhead, because every variable reference is wrapped in a conditional. A simple module like

```
#lang s-exp "racets.rkt"

(define x 10)
(define y 32)
(displayln (+ x y))
```

balloons into

```
#lang s-exp "racets.rkt"

(define x 10)
(define y 32)
(
 ; displayln
 (if (and (box? displayln) (facet? (unbox displayln)))
      (unbox displayln)
      displayln)

  (
   ; +
   (if (and (box? +) (facet? (unbox +)))
        (unbox +)
        +)

    ; x
    (if (and (box? x) (facet? (unbox x)))
      (unbox x)
      x)

    ; y
    (if (and (box? y) (facet? (unbox y)))
      (unbox y)
      y)))
```

Of course, none of these conditionals are actually necessary, because neither `displayln`, nor `+`, nor `x`, nor `y` will ever be boxed facets at runtime in this code.

An abstract interpreter for Racets would have (at a minimum) abstract domains for box and unboxed values, and for faceted and non-faceted values. If the abstract interpreter were able to determine that a particular value was never boxed, or never faceted, then the extra apparatus of faceted execution (like the conditional checks in the code snippet above) could be statically eliminated.

The level of precision that the abstract interpreter attains with its analysis can have a substantial effect on performance. For example, if a certain function is sometimes applied to faceted values and sometimes applied to regular values, the abstract interpreter may or may not be able to identify the call sites that require special handling of the faceted execution, and those that can be run normally. The abstract interpreter's precision is directly related to the implementation of the $\widehat{alloc}$ function: the more distinct addresses that $\widehat{alloc}$ allows, the greater the precision of the analysis, at the cost of slower performance for the abstract interpretation.

Actually deriving an abstract interpreter for Racets (bridging the high-level description in this section with the low-level details in the previous section) is beyond the scope of this undergraduate thesis. See [7] for current work in this area.

5 Conclusion

The primary novel contribution of this thesis is the application of the languages-as-libraries technique to Racets in order to correctly handle certain constructions involving facets and mutable state. Two other issues in the ergonomics of faceted execution have been discussed at some length: type errors and poor performance. I have shown that static typing and abstract interpretation could mitigate these issues.

Further research may proceed in a number of directions. The presentation of static typing and abstract interpretation here has been extremely preliminary, and the details of their concrete implementation in Racets or in other faceted-execution systems remain to be fleshed out. The proof-of-concept of Racets as a language-library will need to be extended with the existing RACETS macros that implement the primitives of faceted execution.

The theory and techniques reviewed in this thesis should contribute to the ergonomics of writing code with faceted execution, and thus in turn to safeguarding the privacy of software users.

Acknowledgements

I would like to thank my thesis advisor, Kristopher Micinski, and my second faculty reader, Steven Lindell, for their valuable feedback and guidance on this project.

References

[1] The Racket reference: fully expanded programs. `https://docs.racket-lang.org/reference/syntax-model.html#(part._fully-expanded)`.

[2] AUSTIN, T. H., YANG, J., AND FLANAGAN, C. Faceted execution of policy-agnostic programs. In *Proceedings of the 8th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security* (2013), ACM, pp. 15–26.

[3] COUSOT, P., AND COUSOT, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of functions. In *Conference Record of the 4th ACM Symposium on the Principles of Programming Languages* (1979), pp. 238–252.

[4] FELLEISEN, M., AND FELLEISEN, D. P. A calculus for assignments in higher-order languages. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (1987).

[5] FELLEISEN, M., AND FRIEDMAN, D. P. Control operators, the SECD-machine, and the lambda-calculus. In *3rd Working Conference on the Formal Description of Programming Concepts* (1986).

[6] HENDERSHOTT, G. Fear of macros. `https://www.greghendershott.com/fear-of-macros/all.html`, June 2018.

[7] MICINSKI, K., DARIS, D., AND GILRAY, T. Abstracting faceted execution: static analysis of dynamic information-flow control for higher-order languages. 2019.

[8] MICINSKI, K., WANG, Z., AND GILRAY, T. RACETS: faceted execution in Racket. 2018.

[9] PIERCE, B. C. *Types and programming languages.* The MIT Press, Cambridge, MA, 2002.

[10] Tobin-Hochstadt, S., St-Amour, V., Culpepper, R., Flatt, M., and Felleisen, M. Languages as libraries. In *Proceedings of the Conference on Programming Language Design and Implementation* (2011), ACM.

[11] Tobin-Hochstadt, S., St-Amour, V., Dobson, E., and Takikawa, A. The Typed Racket guide. `https://docs.racket-lang.org/ts-guide/`.

[12] Van Horn, D., and Might, M. Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, 2010), ACM, pp. 51–62.

[13] Yang, J., Yessenov, K., and Solar-Lezama, A. A language for automatically enforcing privacy policies. *ACM SIGPLAN Notices 47*, 1 (2012), 85–96.