

DrRacket Tools

Version 6.1.1

Robert Bruce Findler

November 4, 2014

This manual describes portions of DrRacket's functionality that are exposed via Racket APIs to be used with other editors.

Contents

1	Accessing Check Syntax Programmatically	3
2	Module Browser	15

1 Accessing Check Syntax Programmatically

```
(require drracket/check-syntax)
package: drracket-tool-lib

(show-content file-or-stx) → (listof vector?)
                          (or/c path-string?
                             file-or-stx : (and/c syntax?
                                              (λ (x) (path-string? (syntax-source x))))))
```

This procedure composes the other pieces of this library together in a way that can be used for REPL-style experimentation with the results from Check Syntax, as shown in the example below. The list it returns has one vector for each call that would be made to the object in `current-annotations`. Each vector's first element is a symbol naming a method in `syncheck-annotations<%>` and the other elements of the vector are the arguments passed to the method.

This doesn't work as well for use in a real tool, however, because it doesn't account for the callback procedures present in `syncheck:add-arrow/name-dup` and `syncheck:add-id-set` and the resulting vectors are probably less convenient to work with than direct method calls for most uses of this library. Nevertheless, it gives a quick feel for the results that can come back from Check Syntax.

See `annotations-mixin` for some example code to use the other parts of this library.

Note that the paths in the example below have been replaced via `make-paths-be-module-paths` in order to make the results be platform independent.

```
> (define (make-paths-be-module-paths x)
  (let loop ([x x])
    (cond
      [(pair? x) (cons (loop (car x)) (loop (cdr x)))]
      [(vector? x) (for/vector ([x (in-vector x)])
                          (loop x))]
      [(path? x)
       (define p (path->collects-relative x))
       (if (path? p)
           (path->main-doc-relative x)
           p)]
      [else x]))))

> (let ([example-module
        '(module m racket (λ (x) x))])
  (make-paths-be-module-paths
   (show-content
    (read-syntax
```

```

        (build-path (current-directory) "dummy-file.rkt")
        (open-input-string (format "~s" example-module))))))
'(#(syncheck:add-require-open-menu 10 16 (collects #"racket"
#"main.rkt"))
#(syncheck:add-tail-arrow 17 26)
#(syncheck:add-arrow/name-dup
  23
  24
  26
  27
  #t
  lexical
  #f
  #<procedure:name-dup?>)
#(syncheck:add-background-color 20 21 "palegreen")
#(syncheck:add-docs-menu
  20
  21
  λ
  "View documentation for λ from racket/base, racket"
  (doc #"reference" #"lambda.html")
  (form ((lib "racket/private/base.rkt") λ))
  "(form._((lib._racket/private/base..rkt)._~ce~bb))")
#(syncheck:add-jump-to-definition
  20
  21
  new-λ
  (collects #"racket" #"private" #"kw.rkt")
  ())
#(syncheck:add-mouse-over-status 20 21 "binding λ imported from
racket")
#(syncheck:add-arrow/name-dup
  10
  16
  20
  21
  #t
  imported
  #t
  #<procedure:name-dup?>)
#(syncheck:add-background-color 20 21 "palegreen")
#(syncheck:add-docs-menu
  20
  21
  λ
  "View documentation for λ from racket/base, racket"

```

```

(doc #reference "lambda.html")
(form ((lib "racket/private/base.rkt") λ))
"(form._((lib._racket/private/base..rkt)._~ce~bb))"
#(syncheck:add-jump-to-definition
  20
  21
  new-λ
  (collects #racket #private #kw.rkt)
  ())
#(syncheck:add-mouse-over-status 20 21 "binding λ imported from
racket")
#(syncheck:add-arrow/name-dup
  10
  16
  20
  21
  #t
  imported
  #t
  #<procedure:name-dup?>)
#(syncheck:add-background-color 1 7 "palegreen")
#(syncheck:add-docs-menu
  1
  7
  module
  "View documentation for module from racket/base, racket"
  (doc #reference "module.html")
  (form ('#%kernel module))
  "(form._((quote._~23~25kernel)._module))"
#(syncheck:add-mouse-over-status 10 16 "1 bound occurrence")
#(syncheck:add-mouse-over-status 23 24 "1 bound occurrence"))

```

```

(namespace? namespace? path) → (→* (syntax?)
                                   ((→ any/c void?))
                                   void?)
                                   (→ void?)
namespace : namespace?
path : (or/c #f path-string?)

```

This function creates some local state about a traversal of syntax objects and returns two functions. The first one should be called with each of the (fully expanded) syntax objects that make up a program (there will be only one if the program is a module) and then the second one should be called to indicate there are no more.

The optional argument to the first function is ignored. It is left there for historical reasons.

In the past it was called for each sequence of binding identifiers encountered in `define-values`, `define-syntaxes`, and `define-values-for-syntax`.

During the dynamic extent of the call to the two result functions, the value of the `current-annotations` parameter is consulted and various methods are invoked in the corresponding object (if any), to indicate what has been found in the syntax object. These methods will only be called if the syntax objects have source locations.

```
(current-annotations)
→ (or/c #f (is-a?/c syncheck-annotations<%>))
(current-annotations ca) → void?
  ca : (or/c #f (is-a?/c syncheck-annotations<%>))
```

The methods of the value of this parameter are invoked by the functions returned from `make-traversal`.

```
(current-max-to-send-at-once)
→ (or/c +inf.0 (and/c exact-integer? (>=/c 2)))
(current-max-to-send-at-once m) → void?
  m : (or/c +inf.0 (and/c exact-integer? (>=/c 2)))
```

No longer used.

```
syncheck-annotations<%> : interface?
```

Classes implementing this interface are acceptors of information about a traversal of syntax objects. See `make-traversal`.

Do not implement this interface directly, as it is liable to change without warning. Instead, use the `annotations-mixin` and override the methods you're interested in. The `annotations-mixin` will keep in sync with this interface, providing methods that ignore their arguments.

```
(send a-syncheck-annotations syncheck:find-source-
object stx)
→ (or/c #f (not/c #f))
  stx : syntax?
```

This should return `#f` if the source of this syntax object is uninteresting for annotations (if, for example, the only interesting annotations are those in the original file and this is a syntax object introduced by a macro and thus has a source location from some other file).

Otherwise, it should return some (non-`#f`) value that will then be passed to one of the other methods below as a `source-obj` argument.

```

(send a-syncheck-annotations syncheck:add-background-color
 source-obj
 start
 end
 color)
→ void?
source-obj : (not/c #f)
start : exact-nonnegative-integer?
end : exact-nonnegative-integer?
color : string?

```

Called to indicate that the color *color* should be drawn on the background of the given range in the editor, when the mouse moves over it. This method is typically called in conjunction with some other method that provides some other annotation on the source.

```

(send a-syncheck-annotations syncheck:add-require-open-menu
 source-obj
 start
 end
 file)
→ void?
source-obj : (not/c #f)
start : exact-nonnegative-integer?
end : exact-nonnegative-integer?
file : path-string?

```

Called to indicate that there is a require at the location from *start* to *end*, and that it corresponds to *file*. Check Syntax adds a popup menu.

```

(send a-syncheck-annotations syncheck:add-docs-menu
 source-obj
 start
 end
 id
 label
 path
 tag)
→ void?
source-obj : (not/c #f)
start : exact-nonnegative-integer?
end : exact-nonnegative-integer?
id : symbol?
label : any/c
path : any/c
tag : any/c

```

Called to indicate that there is something that has documentation between the range *start* and *end*. The documented identifier's name is given by *id* and the docs are found in the html file *path* at the html tag *tag*. The *label* argument describes the binding for use in the menu item (although it may be longer than 200 characters).

```
(send a-syncheck-annotations syncheck:add-id-set
  all-ids
  new-name-interferes?)
→ void?
      (listof (list/c (not/c #f)
                    all-ids : exact-nonnegative-integer?
                          exact-nonnegative-integer?))
new-name-interferes? : (-> symbol boolean?)
```

This method is no longer called by Check Syntax. It is here for backwards compatibility only. The information it provided must now be synthesized from the information supplied to `syncheck:add-arrow/name-dup`.

```
(send a-syncheck-annotations syncheck:add-arrow
  start-source-obj
  start-left
  start-right
  end-source-obj
  end-left
  end-right
  actual?
  phase-level)
→ void?
start-source-obj : (not/c #f)
start-left : exact-nonnegative-integer?
start-right : exact-nonnegative-integer?
end-source-obj : (not/c #f)
end-left : exact-nonnegative-integer?
end-right : exact-nonnegative-integer?
actual? : boolean?
phase-level : (or/c exact-nonnegative-integer? #f)
```

This function is not called directly anymore by Check Syntax. Instead `syncheck:add-arrow/name-dup` is.

This method is invoked by the default implementation of `syncheck:add-arrow/name-dup` in `annotations-mixin`.

```

(send a-syncheck-annotations syncheck:add-arrow/name-dup
 start-source-obj
 start-left
 start-right
 end-source-obj
 end-left
 end-right
 actual?
 phase-level
 require-arrow?
 name-dup?)
→ void?
start-source-obj : (not/c #f)
start-left : exact-nonnegative-integer?
start-right : exact-nonnegative-integer?
end-source-obj : (not/c #f)
end-left : exact-nonnegative-integer?
end-right : exact-nonnegative-integer?
actual? : boolean?
phase-level : (or/c exact-nonnegative-integer? #f)
require-arrow? : boolean?
name-dup? : (-> string? boolean?)

```

Called to indicate that there should be an arrow between the locations described by the first six arguments.

The *phase-level* argument indicates the phase of the binding and the *actual?* argument indicates if the binding is a real one, or a predicted one from a syntax template (predicted bindings are drawn with question marks in Check Syntax).

The *require-arrow?* argument indicates if this arrow points from an imported identifier to its corresponding require.

The *name-dup?* predicate returns *#t* in case that this variable (either the start or end), when replaced with the given string, would shadow some other binding (or otherwise interfere with the binding structure of the program at the time the program was expanded).

```

(send a-syncheck-annotations syncheck:add-tail-arrow
 from-source-obj
 from-pos
 to-source-obj
 to-pos)
→ void?
from-source-obj : (not/c #f)
from-pos : exact-nonnegative-integer?
to-source-obj : (not/c #f)

```

`to-pos` : `exact-nonnegative-integer?`

Called to indicate that there are two expressions, beginning at *from-pos* and *to-pos* that are in tail position with respect to each other.

```
(send a-syncheck-annotations syncheck:add-mouse-over-status
  source-obj
  pos-left
  pos-right
  str)
→ void?
source-obj : (not/c #f)
pos-left : exact-nonnegative-integer?
pos-right : exact-nonnegative-integer?
str : string?
```

Called to indicate that the message in *str* should be shown when the mouse passes over the given position.

```
(send a-syncheck-annotations syncheck:add-jump-to-definition
  source-obj
  start
  end
  id
  filename
  submods)
→ void?
source-obj : (not/c #f)
start : exact-nonnegative-integer?
end : exact-nonnegative-integer?
id : any/c
filename : path-string?
submods : (listof symbol?)
```

Called to indicate that there is some identifier at the given location (named *id*) that is defined in the *submods* of the file *filename* (where an empty list in *submods* means that the identifier is defined at the top-level module).

```
(send a-syncheck-annotations syncheck:add-definition-target
  source-obj
  start
  finish
  style-name)
→ void?
source-obj : (not/c #f)
start : exact-nonnegative-integer?
finish : exact-nonnegative-integer?
style-name : any/c
```

```
(send a-syncheck-annotations syncheck:color-range source-obj
      start
      finish
      style-name
      mode)
→ void?
source-obj : (not/c #f)
start : exact-nonnegative-integer?
finish : exact-nonnegative-integer?
style-name : any/c
mode : any/c
```

Called to indicate that the given location should be colored according to the style *style-name* when in *mode*. The mode either indicates regular check syntax or is used indicate blame for potential contract violations (and still experimental).

```
(send a-syncheck-annotations syncheck:add-rename-menu
      id
      all-ids
      new-name-interferes?)
→ void?
id : symbol?
      (listof (list/c (not/c #f)
                    exact-nonnegative-integer?
                    exact-nonnegative-integer?))
all-ids :
new-name-interferes? : (-> symbol boolean?)
```

This method is listed only for backwards compatibility. It is not called by Check Syntax anymore.

```
annotations-mixin : (class? . -> . class?)
result implements: syncheck-annotations<%>
```

Supplies all of the methods in `syncheck-annotations<%>` with default behavior. Be sure to use this mixin to future-proof your code and then override the methods you're interested in.

By default:

- The `syncheck:find-source-object` method ignores its arguments and returns `#f`;
- the `syncheck:add-arrow/name-dup` method drops the `require-arrow?` and `name-dup?` arguments and calls `syncheck:add-arrow`; and
- all of the other methods ignore their arguments and return `(void)`.

Here is an example showing how use this library to extract all of the arrows that Check Syntax would draw from various expressions. One subtle point: arrows are only included when the corresponding identifiers are `syntax-original?`; the code below manages this by copying the properties from an identifier that is `syntax-original?` in the call to `datum->syntax`.

```

> (define arrows-collector%
  (class (annotations-mixin object%)
    (super-new)
    (define/override (syncheck:find-source-object stx)
      stx)
    (define/override (syncheck:add-arrow/name-dup
      start-source-obj start-left start-right
      end-source-obj end-left end-right
      actual? phase-level require-arrow? name-
dup?)
      (set! arrows
        (cons (list start-source-obj end-source-obj)
              arrows)))
    (define arrows '())
    (define/public (get-collected-arrows) arrows)))

> (define (arrows form)
  (define base-namespace (make-base-namespace))
  (define-values (add-syntax done)
    (make-traversal base-namespace #f))
  (define collector (new arrows-collector%))
  (parameterize ([current-annotations collector]
                 [current-namespace base-namespace])
    (add-syntax (expand form))
    (done))
  (send collector get-collected-arrows))

> (define (make-id name pos orig?)
  (datum->syntax
   #f
   name
   (list #f #f #f pos (string-length (symbol->string name))))
  (and orig? #'is-orig)))

> (arrows `(λ (,(make-id 'x 1 #t)) ,(make-id 'x 2 #t)))
'((#<syntax::1 x> #<syntax::2 x>))
> (arrows `(λ (x) x))
'()
> (arrows `(λ (,(make-id 'x 1 #f)) ,(make-id 'x 2 #t)))

```

```
'()
> (arrows `(lambda (,(make-id 'x 1 #t)) x))
'()
```

| `syncheck:find-source-object`

Bound to an identifier created with `define-local-member-name` that is used in `syncheck-annotations<%>`.

| `syncheck:add-background-color`

Bound to an identifier created with `define-local-member-name` that is used in `syncheck-annotations<%>`.

| `syncheck:add-require-open-menu`

Bound to an identifier created with `define-local-member-name` that is used in `syncheck-annotations<%>`.

| `syncheck:add-docs-menu`

Bound to an identifier created with `define-local-member-name` that is used in `syncheck-annotations<%>`.

| `syncheck:add-rename-menu`

Bound to an identifier created with `define-local-member-name` that is used in `syncheck-annotations<%>`.

| `syncheck:add-arrow`

Bound to an identifier created with `define-local-member-name` that is used in `syncheck-annotations<%>`.

| `syncheck:add-arrow/name-dup`

Bound to an identifier created with `define-local-member-name` that is used in `syncheck-annotations<%>`.

`syncheck:add-tail-arrow`

Bound to an identifier created with `define-local-member-name` that is used in `syncheck-annotations<%>`.

`syncheck:add-mouse-over-status`

Bound to an identifier created with `define-local-member-name` that is used in `syncheck-annotations<%>`.

`syncheck:add-jump-to-definition`

Bound to an identifier created with `define-local-member-name` that is used in `syncheck-annotations<%>`.

`syncheck:add-id-set`

Bound to an identifier created with `define-local-member-name` that is used in `syncheck-annotations<%>`.

`syncheck:color-range`

Bound to an identifier created with `define-local-member-name` that is used in `syncheck-annotations<%>`.

2 Module Browser

```
(require drracket/module-browser)  
package: drracket-tool-lib
```

```
(module-browser path) → void?  
  path : path-string?
```

Opens a window containing the module browser for *path*.