Macro Debugger: Inspecting Macro Expansion

Version 5.1.3

Ryan Culpepper

August 15, 2011

The macro-debugger collection contains two tools: a stepper for macro expansion and a standalone syntax browser. The macro stepper shows the programmer the expansion of a program as a sequence of rewriting steps, using the syntax browser to display the individual terms. The syntax browser uses colors and a properties panel to show the term's syntax properties, such as lexical binding information and source location.

1 Macro stepper

```
(require macro-debugger/stepper)
```

(expand/step *stx*) → void?
  stx : any/c

Expands the syntax (or S-expression) and opens a macro stepper frame for stepping through the expansion.

(expand-module/step *mod*) → void?
  mod : module-path?

Expands the source file named by *mod*, which must contains a single module declaration, and opens a macro stepper frame for stepping through the expansion.

2 Macro expansion tools

```
(require macro-debugger/expand)
```

This module provides `expand`-like procedures that allow the user to specify macros whose expansions should be hidden.

Warning: because of limitations in the way macro expansion is selectively hidden, the resulting syntax may not evaluate to the same result as the original syntax.

```
(expand-only stx transparent-macros) → syntax?
  stx : any/c
  transparent-macros : (listof identifier?)
```

Expands the given syntax *stx*, but only shows the expansion of macros whose names occur in *transparent-macros*.

Example:

```
> (syntax->datum
   (expand-only #'(let ([x 1] [y 2]) (or (even? x) (even? y)))
                (list #'or)))
'(let ((x 1) (y 2)) (let ((or-part (even? x))) (if or-part or-part
(even? y))))
```

```
(expand/hide stx hidden-macros) → syntax?
  stx : any/c
  hidden-macros : (listof identifier?)
```

Expands the given syntax *stx*, but hides the expansion of macros in the given identifier list (conceptually, the complement of expand-only).

Example:

```
> (syntax->datum
   (expand/hide #'(let ([x 1] [y 2]) (or (even? x) (even? y)))
                (list #'or)))
'(let-values (((x) '1) ((y) '2)) (or (#%app even? x) (#%app even?
y)))
```

```
(expand/show-predicate stx show?) → syntax?
  stx : any/c
  show? : (-> identifier? boolean?)
```

Expands the given syntax *stx*, but only shows the expansion of macros whose names satisfy the predicate *show?*.

Example:

```
> (syntax->datum
   (expand/show-predicate
    #'(let ([x 1] [y 2]) (or (even? x) (even? y)))
    (lambda (id) (memq (syntax-e id) '(or #%app)))))
'(let ((x 1) (y 2)) (let ((or-part (#%app even? x))) (if or-part
or-part (#%app even? y))))
```

3 Macro stepper API for macros

```
(require macro-debugger/emit)
```

Macros can explicitly send information to a listening macro stepper by using the procedures in this module.

```
(emit-remark fragment ... [#:unmark? unmark?]) → void?
  fragment : (or/c syntax? string?)
  unmark? : boolean? = #t
```

Emits an event to the macro stepper (if one is listening) containing the given strings and syntax objects. The macro stepper displays a remark by printing the strings and syntax objects above a rendering of the macro's context. The remark is only displayed if the macro that emits it is considered transparent by the hiding policy.

By default, syntax objects in remarks have the transformer's mark applied (using `syntax-local-introduce`) so that their appearance in the macro stepper matches their appearance after the transformer returns. Unmarking is suppressed if *unmark?* is `#f`.

```
(define-syntax (mymac stx)
  (syntax-case stx ()
    [(_ x y)
     (emit-remark "I got some arguments!"
                  #'x
                  "and"
                  #'y)
     #'(list 'x 'y)]))
(mymac 37 (+ 1 2))
```

(Run the fragment above in the macro stepper.)

```
(emit-local-step before after #:id id) → void?
  before : syntax?
  after : syntax?
  id : identifier?
```

Emits an event that simulates a local expansion step from *before* to *after*.

The *id* argument acts as the step's "macro" for the purposes of macro hiding.

4 Macro stepper text interface

```
(require macro-debugger/stepper-text)
```

```
(expand/step-text stx [show?]) → void?
  stx : any/c
  show? : (or/c (-> identifier? boolean?) = (lambda (x) #t)
                (listof identifier?))
```

Expands the syntax and prints the macro expansion steps. If the identifier predicate is given, it determines which macros are shown (if absent, all macros are shown). A list of identifiers is also accepted.

Example:

```
> (expand/step-text #'(let ([x 1] [y 2]) (or (even? x) (even? y)))
                     (list #'or))
Macro transformation
(let ((x 1) (y 2)) (or (even? x) (even? y)))
  ==>
(let
 ((x 1) (y 2))
 (let:1 ((or-part:1 (even? x))) (if:1 or-part:1 or-part:1 (or:1
(even? y)))))

Macro transformation
(let
 ((x 1) (y 2))
 (let:1 ((or-part:1 (even? x))) (if:1 or-part:1 or-part:1 (or:1
(even? y)))))
  ==>
(let
 ((x 1) (y 2))
 (let:1
  ((or-part:1 (even? x)))
  (if:1 or-part:1 or-part:1 (#%expression:2 (even? y)))))

Macro transformation
(let
 ((x 1) (y 2))
 (let:1
  ((or-part:1 (even? x)))
  (if:1 or-part:1 or-part:1 (#%expression:2 (even? y)))))
  ==>
(let
```

```
  ((x 1) (y 2))
  (let:1 ((or-part:1 (even? x))) (if:1 or-part:1 or-part:1 (even?
 y))))
```

```
(stepper-text stx [show?]) → (symbol? -> void?)
  stx : any/c
  show? : (or/c (-> identifier? boolean?) = (lambda (x) #t)
                (listof identifier?))
```

Returns a procedure that can be called on the symbol `'next` to print the next step or on the symbol `'all` to print out all remaining steps.

5 Syntax browser

```
(require macro-debugger/syntax-browser)
```

(browse-syntax *stx*) → void?
  stx : syntax?

Creates a frame with the given syntax object shown. More information on using the GUI is available below.

(browse-syntaxes *stxs*) → void?
  stxs : (listof syntax?)

Like browse-syntax, but shows multiple syntax objects in the same frame. The coloring partitions are shared between the two, showing the relationships between subterms in different syntax objects.

6 Using the macro stepper

6.1 Navigation

The stepper presents expansion as a linear sequence of rewriting process, and it gives the user controls to step forward or backwards as well as to jump to the beginning or end of the expansion process.

If the macro stepper is showing multiple expansions, then it also provides “Previous term” and “Next term” buttons to go up and down in the list of expansions. Horizontal lines delimit the current expansion from the others.

6.2 Macro hiding

Macro hiding lets one see how expansion would look if certain macros were actually primitive syntactic forms. The macro stepper skips over the expansion of the macros you designate as opaque, but it still shows the expansion of their subterms.

The bottom panel of the macro stepper controls the macro hiding policy. The user changes the policy by selecting an identifier in the syntax browser pane and then clicking one of “Hide module”, “Hide macro”, or “Show macro”. The new rule appears in the policy display, and the user may later remove it using the "Delete" button.

The stepper also offers coarser-grained options that can hide collections of modules at once. These options have lower precedence than the rules above.

Macro hiding, even with no macros marked opaque, also hides certain other kinds of steps: internal defines are not rewritten to letrecs, begin forms are not spliced into module or block bodies, etc.

7 Using the syntax browser

7.1 Selection

The selection is indicated by bold text.

The user can click on any part of a subterm to select it. To select a parenthesized subterm, click on either of the parentheses. The selected syntax is bolded. Since one syntax object may occur inside of multiple other syntax objects, clicking on one occurrence will cause all occurrences to be bolded.

The syntax browser displays information about the selected syntax object in the properties panel on the right, when that panel is shown. The selected syntax also determines the highlighting done by the secondary partitioning (see below).

7.2 Primary partition

The primary partition is indicated by foreground color.

The primary partitioning always assigns two syntax subterms the same color if they have the same marks. In the absence of unhygienic macros, this means that subterms with the same foreground color were either present in the original pre-expansion syntax or generated by the same macro transformation step.

Syntax colored in black always corresponds to unmarked syntax. Such syntax may be original, or it may be produced by the expansion of a nonhygienic macro.

Note: even terms that have the same marks might not be `bound-identifier=?` to each other, because they might occur in different environments.

7.3 Secondary partitioning

The user may select a secondary partitioning through the Syntax menu. This partitioning applies only to identifiers. When the user selects an identifier, all terms in the same equivalence class as the selected term are highlighted in yellow.

The available secondary partitionings are:

- `bound-identifier=?`
- `free-identifier=?`

7.4 Properties

When the properties pane is shown, it displays properties of the selected syntax object. The properties pane has two tabbed pages:

- **Term**:

  If the selection is an identifier, shows the binding information associated with the syntax object. For more information, see `identifier-binding`, etc.

- **Syntax Object**:

  Displays source location information and other properties (see `syntax-property`) carried by the syntax object.

7.5 Interpreting syntax

The binding information of a syntax object may not be the same as the binding structure of the program it represents. The binding structure of a program is only determined after macro expansion is complete.

8 Checking requires

```
(require macro-debugger/analysis/check-requires)
```

```
(check-requires module-name)
 → (listof (list/c 'keep   module-path-index? number? (or/c string? #f))
           (list/c 'bypass module-path-index? number?)
           (list/c 'drop   module-path-index? number?))
  module-name : module-path?
```

Estimate a module's useless requires. The procedure returns one element per (non-label) require in the following format:

- `'keep` `module` at `phase` `(optional-comment)`
  - The require must be kept because bindings defined within it are used.
  - The optional comment indicates if the require must be kept
    - only because its bindings are re-exported
    - only because the whitelist DB says so
- `'bypass` `module` at `phase`
  - The require is used, but only for bindings that could be more directly obtained via another module. For example, `racket` can be bypassed in favor of some subset of `racket/base`, `racket/contract`, etc.
- `'drop` `module` at `phase`
  - The require appears to be unused. Unless it must be kept for side effects or for bindings of a very unusual macro, it can be dropped entirely.

Examples:

```
(check-requires 'typed-scheme)
(check-requires 'unstable/markparam)
(check-requires 'macro-debugger/syntax-browser/widget)
```

A scripting interface to `macro-debugger/analysis/check-requires` usable from the command-line is available at `macro-debugger/analysis/check-requires-script.rkt`.

Example (from racket root directory):

```
  racket -l macro-debugger/analysis/check-requires-script \
collects/syntax/*.rkt
```

```
(show-requires module-name)
 → (listof (list/c 'keep   module-path? number? (or/c string? #f))
           (list/c 'bypass module-path? number?)
           (list/c 'drop   module-path? number?))
  module-name : module-path?
```

Similar to `check-requires`, but outputs module paths instead of module path indexes, for more readability.