Interpreter Events

Table of Contents

1. Concept

In order to support writing debuggers, profilers, analyzers and other useful tools that are needed for efficient programming, the interpreter is offering a generic event-oriented low-level API on top of which all these tools can be built. It is similar to the `parse/trace` and `lexer/trace` instrumentation approach.

2. Interpreter instrumentation

In order to access internal interpreter states, the interpreter is generating events at key points of his internal code evaluation process. Those events can be captured using a user-provided callback function. Events are emitted only if a *tracing* mode is enabled in the interpreter using `/trace` refinement on `do`.

```
do/trace <code> <callback>

<code>     : a block of code to evaluate in tracing mode.
<callback> : a callback function, triggered on each interpreter event.
```

Example

```
logger: function [
    event  [word!]                      ;-- Event name
    code   [any-block! none!]           ;-- Currently evaluated block
    offset [integer!]                   ;-- Offset in evaluated block
    value  [any-type!]                  ;-- Value currently processed
    ref    [any-type!]                  ;-- Reference of current call (usually word or
path)
    frame  [pair!]                      ;-- Stack frame start/top positions
][
    print [
        pad uppercase form event 8
        mold/part/flat either any-function? :value [:ref][:value] 20
    ]
]

do/trace [print 1 + 2] :logger
```

will output:

```
INIT    none                    ;-- Initializing tracing mode
ENTER   none                    ;-- Entering block to evaluate
FETCH   print                   ;-- Fetching and evaluating `print` value
OPEN    print                   ;-- Results in opening a new call stack frame
FETCH   +                       ;-- Fetching and evaluating `+` infix operator
OPEN    +                       ;-- Results in opening a new call stack frame
FETCH   1                       ;-- Fetching left operand `1`
PUSH    1                       ;-- Pushing integer! value `1` on stack
FETCH   2                       ;-- Fetching and evaluating right operand
PUSH    2                       ;-- Pushing integer! value `2`
CALL    +                       ;-- Calling `+` operator
RETURN  3                       ;-- Returning the resulting value
CALL    print                   ;-- Calling `print`
3                               ;-- Outputting 3
RETURN  unset                   ;-- Returning the resulting value
EXIT    none                    ;-- Exiting evaluated block
END     none                    ;-- Ending tracing mode
```

3. Events

When the tracing mode is active, the interpreter will trigger events described below. Events can be grouped into the following categories:

- Global events: `INIT`, `END`
- Evaluating a block/paren/path of code: `ENTER`, `EXIT`
- Calling any type of function: `OPEN`, `CALL`, `RETURN`
- Evaluating a function body block: `PROLOG`, `EPILOG`
- Expression evaluation: `FETCH`, `PUSH` , `SET`, `EXPR`, `ERROR`
- Exceptions handling: `THROW`, `CATCH`

Detailed description:

Event	Code	Offset	Value	Ref	Description
`INIT`	`none`	-1	n/a (`none`)	n/a (`none`)	when the tracing mode is initiated (`do/trace` call).
`END`	`none`	-1	n/a (`none`)	n/a (`none`)	when the tracing mode is ended (`do/trace` call exiting).
`ENTER` [(1)]	`any-block!`	-1	n/a (`none`)	n/a (`none`)	when a block is about to be evaluated.
`EXIT` [(1)]	`any-block!`	-1	last `any-type!` evaluated value	n/a (`none`)	when current evaluated block's or path's tail has been reached.
`OPEN`	`any-block!`	`integer!`	`any-function!`	`word!`, `path!`	when a new function (any-function!) call is initiated and a new stack frame is opened.
`CALL`	`any-block!`	`integer!`	`any-function!` to call	`word!`, `path!`, `any-function!`	a function with all arguments fetched on the stack gets called.
`RETURN`	`any-block!`	`integer!`	returned `any-type!` value	`word!`, `path!`	when a function call has returned and its stack frame has been closed.
`EXPR`	`any-block!`	`integer!`	expression `any-type!` result	n/a (`none`)	when a top-level expression has been evaluated.
`PROLOG`	`any-block!`	-1	called `function!` value	`word!`, `path!`	when entering a function! body.

Event	Code	Offset	Value	Ref	Description
EPILOG	any-block!	-1	called function! value	word!, path!	when exiting a function! body.
FETCH	any-block!	integer!	fetched any-type! value	n/a (none)	a value is read from the input block to be evaluated.
PUSH	any-block!	integer!	pushed any-type! value	n/a (none)	a value has been pushed on the stack.
SET	any-block!	integer!	any-type!	set-word!, set-path!	a set-word or set-path is set to a value.
ERROR	none	-1	error! value	n/a (none)	when an error occurs and is about to be thrown up in the stack.
THROW	none	-1	thrown any-type! value	n/a (none)	when a value is thrown using throw native.
CATCH	none	-1	thrown any-type! value	n/a (none)	when a value is caught using catch native.

[1] Note that a pair of enter and exit events are generated for any path evaluation (like for other block datatypes).

Events come with extra information:

- code: when available, it provides the input block! or paren! series currently interpreted.
- offset: when different from -1, indicates the input series offset at the event moment.
- value: when available, the currently processed value.
- ref: when available, references the word or path from which evaluation produced the current event/value.

4. Event handler

Here is the prototype of event handlers suitable to be passed as argument to do/trace:

```
func [
    event  [word!]
    code   [any-block! none!]
    offset [integer!]
    value  [any-type!]
    ref    [any-type!]
    frame  [pair!]
][
    [events]                    ;-- optional restricted event names list
    ...body...
]
```

Argument	Description
`event`	Event name.
`code`	Block of code currently evaluated.
`offset`	Offset in block currently evaluated.
`value`	Value currently processed in the event.
`ref`	Reference of the call (word or path) associated to the event.
`frame`	Pair of indexes in the Red internal stack denoting the beginning and end of the call frame. [(1)]

[(1)] Note that the `frame` index range is for the internal Red stack, not the one used in the debugger (which is managed by the debugger itself).

The body block can start with an optional filtering block, for indicating which events will be triggered. This allows to reduce the number of callback calls resulting in much better processing performance.

5. Tracing control

5.1. `trace` function

Syntax

```
    trace <mode>

    <mode>: new event generation mode (logic!)
```

Description

`trace` allows to turn on/off event generation during a traced evaluation (inside code evaluated with `do/trace`). Using `trace` in such way outside of a traced evaluation has no effect. `trace` function has also another usage described in "Predefined tools" section.

Example

```
do/trace [
    print "before"
    trace off
    print "between"
    trace on
    print "after"
] :logger
```

will output

```
INIT     none
ENTER    none
FETCH    print
OPEN     print
FETCH    "before"
PUSH     "before"
CALL     print
before
RETURN   unset
FETCH    trace
OPEN     trace
FETCH    off
PUSH     false
CALL     trace                ;-- calling `trace off`
between                       ;-- only `print` output but no related events
RETURN   true                 ;-- next event is the return of `trace on`
FETCH    print
OPEN     print
FETCH    "after"
PUSH     "after"
CALL     print
after
RETURN   unset
EXIT     unset
END      none
```

5.2. Function attributes

Any function called during a traced evaluation can be set to either avoid generating any event or be forced to generate events while event generation is disabled. This can be achieved using the following function attributes.

5.2.1. trace

Syntax

```
    func [[trace]...][...]
```

Description

When this attribut is used, the function will be forced to generate events during a traced evaluation. This propagates to nested calls also (unless they explicitly disable event generation). Note that it is still possible to locally turn event generation on/off using `trace`.

Example

```
foo: func [[trace]][1 + 2]
do/trace [
    trace off
    print "before"
    foo
    print "after"
    trace on
] :logger
```

will output

```
INIT     none
ENTER    none
FETCH    trace
OPEN     trace
FETCH    off
PUSH     false
CALL     trace
before                          ;-- no related events for `print "before"`
PROLOG   foo                    ;-- events enabled from beginning of `foo` body
evaluation
ENTER    none
FETCH    +
OPEN     +
FETCH    1
PUSH     1
FETCH    2
PUSH     2
CALL     +
RETURN   3
EXIT     3
EPILOG   foo                    ;-- event generation stopped again when `foo` exits
after                           ;-- no related events for `print "after"`
RETURN   true                   ;-- next event is the return of `trace on`
EXIT     true
END      none
```

5.2.2. no-trace

Syntax

```
    func [[no-trace]...][...]
```

Description

When this attribut is used, the function will be blocked from generating events during a traced evaluation. This propagates to nested calls also (unless they explicitly disable event generation). Note that it is still possible to locally turn event generation on/off using `trace`.

Example

```
foo: func [[no-trace]][print 1 + 2]
do/trace [print "before" foo print "after"] :logger
```

will output

```
INIT     none
ENTER    none
FETCH    print
OPEN     print
FETCH    "before"
PUSH     "before"
CALL     print
before
RETURN   unset
FETCH    foo
OPEN     foo
CALL     foo                    ;-- last event before entering `foo`
3                               ;-- no event generated from inside `foo`
RETURN   unset                  ;-- next event is the return from `foo`
FETCH    print
OPEN     print
FETCH    "after"
PUSH     "after"
CALL     print
after
RETURN   unset
EXIT     unset
END      none
```

6. Predefined tools

Several handlers are always available in the Red runtime library in order to help users better

analyze and debug Red programs.

6.1. debug

Syntax

```
debug <code>
debug/later <code>

<code> : code to evaluate through the debugger (any-type!)
```

Description

Starts an interactive debugging session, allowing to evaluate the `code` argument in a controlled way. A debugging console is presented using the `debug>_` prompt, waiting for user commands (see the list below).

The `/later` refinement will let the evaluation run uninterrupted until a `@stop` value is encountered, entering the debugging console. The normal evaluation can be resumed when encountering the `@go` value (in addition to the `continue` debug command). The `@stop` value acts effectively as a breakpoint. Using the `@stop` and `@go` values allows to selectively enter the step by step evaluation, only on chosen code pieces.

Debugger command list:

Command	Shortcut	Description
`help`	`?`	Print a list of debugger's commands.
`next`	`n` or ENTER key	Next evaluation step.
`continue`	`c`	Exit debugging console but continue evaluation.
`quit`	`q`	Exit debugging console and stop evaluation.
`stack`	`s`	Display the current local calls and expression stack.
`parents`	`p`	Display the parents call stack.
`:word`	n/a	Outputs the value of `word`. If it is a `function!`, outputs the local context.
`:a/b/c`	n/a	Outputs the value of `a/b/c` path.
`watch <word1> <word2>⋯`	`w ⋯`	Watch one or more words.
`-watch <word1> <word2>⋯`	`-w ⋯`	Stop watching one or more words.
`+stack`	`+s`	Outputs expression stack on each new event.
`-stack`	`-s`	Do not output expression stack on each new event.

Command	Shortcut	Description
`+locals`	`+l`	Output local context for each entry in the call stack.
`-locals`	`-l`	Do not output local context for each entry in the call stack.
`+indent`	`+i`	Indent the output of the expression stack.
`-indent`	`-i`	Do not indent the output of the expression stack.

When the interactive mode is entered, the debugger console will output a set of contextual information on every step. Here is a short description:

Typical output:

```
debug> n                  ;-- last debug command (`next`)
-----> EVAL n             ;-- the next step: fetching and evaluating `n`
Input: n < 1              ;-- the currently evaluated code
Stack: print              ;-- the current (local) call stack bottom
Stack: fibo               ;-- both function calls and their arguments are shown on
stack
Stack: 4
Stack: either
Stack: <                  ;-- stack top
```

That stack is referred to as the "expression stack" to contrast it with the "parents stack" which refers to the call stack above the point when the debugger was called. When the expression stack is empty, an `-empty stack-` label is displayed under the `Input:` line.

6.2. `profile`

Syntax

```
    profile <code>
    profile/by <code> <category>

    <code>     : code to profile (any-type!)
    <category> : sort by a specific category: 'name, 'count, 'time (word!)
```

Description

Profiles the provided code, counting function invocations and measuring duration. Once the code evaluation returns, a report is printed. The default sorting is per invocation count. Alternative sorting can be used through the `/by` refinement. `profile` accepts the same arguments as `do`.

Notes:

- Timing is currently not very accurate for durations less than 20ms on Windows platform

(default timer accuracy). This will be improved in the future with better timers and functions prolog/epilog more accurate exclusions.

- Nested functions duration are currently added to their parent timing. Proper function timing (excluding nested calls) will be added in the future.
- Function calls with refinement are counted separately as specfic function instances (same refinements in different order will be counted separately too currently).

Options

By default, `profile` will account for any type of functions (`any-function!` typeset). It is possible to restrict to a sub-group by directly modifying the option `system/tools/options/profile/types`, setting it to a different typeset.

Examples

```
profile [print 1 + 2 + 3 * 5]

30
#1   +               | 2          | 0:00:00
#2   *               | 1          | 0:00:00
#3   print           | 1          | 0:00:00.001
```

Files and URLs can be passed directly as argument:

```
profile https://raw.githubusercontent.com/red/red/master/tests/demo.red

        RedRed              d
        d     d             e
        e     e             R
        R     R   edR    dR d
        d     d  d   R  R  Re
        edRedR   e   d  d   R
        R   e    RedR   e   d
        d    e   d      R   e
        e    R   e   d  d  dR
        R     R   edR    dR d
#1   if               | 420        | 0:00:00
#2   <=               | 391        | 0:00:00
#3   prin             | 241        | 0:00:00.240773
#4   +                | 220        | 0:00:00
#5   either           | 210        | 0:00:00
#6   all              | 210        | 0:00:00.0028192
#7   >                | 210        | 0:00:00.0020021
#8   =                | 210        | 0:00:00.0010021
#9   tail?            | 37         | 0:00:00
#10  unless           | 37         | 0:00:00
#11  skip             | 37         | 0:00:00
#12  repeat           | 10         | 0:00:00.212984
#13  next             | 10         | 0:00:00
#14  foreach          | 1          | 0:00:00.251109
```

Options can be modified ahead of time to change the set of function types processed:

```
system/tools/options/profile/types: make typeset! [op!]
profile https://raw.githubusercontent.com/red/red/master/tests/demo.red

        RedRed              d
        d     d             e
        e     e             R
        R     R   edR    dR d
        d     d  d   R  R  Re
        edRedR   e   d  d   R
        R   e    RedR   e   d
        d    e   d      R   e
        e    R   e   d  d  dR
        R     R   edR    dR d
#1   <=               | 391        | 0:00:00.0000038
#2   +                | 220        | 0:00:00
#3   >                | 210        | 0:00:00
#4   =                | 210        | 0:00:00.0010005
```

6.3. trace

Syntax

```
    trace <code>
    trace <mode>
    trace/raw <code>

    <code> : code to trace (any-type!)
    <mode> : turn tracing on/off (logic!)
```

Description

Generates a simple trace of the argument evaluation steps. Only the following subset of all the possible interpreter events will be shown: `open push call prolog epilog set return error catch throw`. In order to display a lower level trace with all the events, a `/raw` refinement is provided. The output then just dumps the following information for each event: event name, offset, reference, value, frame range (basically the event handler arguments, except for the `code` argument).

When a `logic!` value is passed as argument to `trace`, it will just switch the tracing on/off, allowing a tigher control from within a traced code evaluation.

Options

By default, the output trace will be indented on nested calls. It is possible to make the trace "flat" by setting the option `system/tools/options/trace/indent?` to `false`.

Examples

```
trace [a: 1 + 2]
```

will output

```
-> PUSH a:
-> OPEN +
->  PUSH 1
->  PUSH 2
->  CALL op!  (+)
-> RETURN 3  (+)
-> SET 3  (a)
== 3
```

Using the `/raw` refinement:

```
trace/raw [a: 1 + 2]
```

will output

```
INIT -1 none none 36x38
ENTER 0 none none 38x38
FETCH 0 none a: 38x38
PUSH 0 none a: 38x39
FETCH 1 none + 38x39
OPEN 1 none + 38x39
FETCH 1 none 1 39x39
PUSH 1 none 1 39x40
FETCH 3 none 2 39x40
PUSH 3 none 2 39x41
CALL 4 + make op! [["Returns the sum of 39x41
RETURN 4 + 3 39x41
SET 4 a: 3 38x40
EXPR 4 none 3 38x39
EXIT 4 none 3 38x39
END -1 none none 36x39
== 3
```

6.4. `tracing?`

Syntax

```
    <state>: tracing?

    <state> : returns the current event generation mode (logic!)
```

Description

Reports the state of the current interpreter event generation (`true` or `false`).

Example

```
foo: func [[no-trace]][probe tracing? print 1 + 2]
no-log: func [e c o v r f][]
do/trace [probe tracing? foo probe tracing?] :no-log
```

will output

```
true
false
3
true
```

7. Implementation notes

7.1. General notes

minimal code

7.2. Profiler improvements

7.3. Stack access