Sandbox Primitives

Aaron W. Hsu <arcfide@sacrideo.us>
Karissa R. McKelvey <krmckelv@indiana.edu>

2 March 2011

Contents

Library Definition: (mags sandbox)

This library contains various procedures for constructing a sandbox environment. This sandbox is in essence a safe testing environment for which we can do multiple things:

- Evaluate expressions in a safe environment.
- Identify a procedure that could use an expression that would have have implications we want to avoid. We can `flag` these as expressions. You can see how this works in the section titled Creating Sandboxes.
- Identify an infinite loop or inefficient procedure.

Exports:

`run-sandbox-tests`	`timeout?`	`illegal-term-name`
`&sandbox`	`illegal-term`	`eval/sandbox`
`&timeout`	`illegal-term?`	`load/sandbox`
`&illegal-term`	`sandbox-i/o-error`	`library/sandbox`
`&sandbox-i/o-error`	`sandbox-i/o-error?`	`flag`
`&unbound-term`	`unbound-term`	`import`
`sandbox-condition`	`unbound-term?`	`sandbox`
`sandbox-condition?`	`timeout-engine`	
`timeout`	`sandbox-i/o-error-port`	

Imports:

```
(chezscheme) (srfi :64)
```

1. Overview. THis is an overview for the sandbox environment

2. Condition Hierarchy. These conditions are raised when certain problems arise in the evaluation of terms inside the sandbox environment. There are four conditions as follows:

- `&sandbox` (the parent of all subsequent conditions)
- `&timeout`
- `&sandbox-i/o-error`
- `&illegal-term`

Each has its own special occasion for use, as well as constructors, accessors, and predicates.

`&sandbox` is the parent condition for all of the following sandbox conditions. The constructor is `sandbox-condition` and predicate `sandbox-condition?`. It takes no arguments.

⟨∗⟩ ≡

```
(define-condition-type
  &sandbox
  &condition
  sandbox-condition
  sandbox-condition?)
```

3. `&timeout` is a child condition of the parent `&sandbox`. This condition is raised when the evaluation of a certain procedure runs out of time. It takes one argument, `timeout-engine` which represents the unique engine from which this timeout condition was raised. Predicate: `timeout?`, Constructor: `(timeout timeout-engine)`.

⟨∗⟩ ≡

```
(define-condition-type &timeout &sandbox timeout timeout?
  (engine timeout-engine))
```

4. `&sandbox-i/o-error` is a child condition of the parent type `&sandbox`. This condition is raised when there arises an input/output error or filesystem error when the sandbox loads or writes files. It takes one argument, `sandbox-i/o-error-port`, i.e. the port where the error occured. Predicate: `sandbox-i/o-error?`, Constructor: `(sandbox-i/o-error port)`.

⟨∗⟩ ≡

```
(define-condition-type &sandbox-i/o-error &sandbox sandbox-i/o-error
```

```
  sandbox-i/o-error?  (port sandbox-i/o-error-port))
```

5. `&illegal-term` is a child condition of the parent type `&sandbox`, This conditions is raised when there arises a term deemed illegal that is evaluated in the sandbox. It takes one argument `illegal-term-name` that is the term deemed illegal by the sandbox. Predicate: `illegal-term?`, Constructor: `(illegal-term name)`.

⟨∗⟩ ≡

```
(define-condition-type &illegal-term &sandbox illegal-term
  illegal-term?  (name illegal-term-name))
```

6. `&unbound-term` is a child condition of the parent `&sandbox`. This condition is raised when the evaluation of a procedure expected to be found is not found in the code within the sandbox. It takes one argument, the `unbound-term-name`, the name of the term which was determined to be unbound. Predicate: `unbound-term?`, Constructor: `(unbound-term name)`.

⟨∗⟩ ≡

```
(define-condition-type &unbound-term &sandbox unbound-term
  unbound-term?  (name unbound-term-name))
```

7. Creating sandboxes. Sandboxes are just libraries, but there are certain expectations about sandboxes that suggest another interface to create them. For example, sandboxes generally do not define new bindings; instead, most of the time they define some subset of other libraries of code.

In our particular take on sandboxes, we may want the use of certain terms (that are normally defined), such as `append`, to raise an error as soon as we realize they are being used. We say that these names are flagged. We have the following syntax to make these things easier:

```
(library/sandbox name
  (import imp-spec ...)
  (flag id ...))
```

Here, `imp-spec ...` is a list of any normal R6RS import clause. The `id ...` list is simply a list of identifiers. The result of evaluating the above code is a newly defined library called `name` which exports all the bindings exported by the specifications `imp-spec ...` as well as a set of identifier syntaxes `id ...` that, if invoked, raise an `&illegal-term` condition. The identifiers in the flag clause must not be exported by any `imp-spec ...` clause.

The most interesting aspect of this code is extracting the set of exports from a set of import specifications. This is accomplished using environments.

⟨Compute export list⟩ ≡

```
(datum->syntax
  #'k
  (environment-symbols
    (apply environment (syntax->datum #'(imp (...  ...))))))
```

Captures: `k imp`

8. Evaluating code within Sandboxes. These procedures aid in the evaluation of expressions or files within a given environment-name with a given amount of time in order to spot an infinite loop or inefficient proceure. Using the following code, we run an engine which returns the value of the expression unless the given time runs to zero, in which case it will create and raise a `&timeout` condition with the expression name.

⟨Run Timeout Engine⟩ ≡

```
(eng time
     (lambda (ticks value) value)
     (lambda (x) (raise-continuable (timeout x))))
```

Captures: `eng time`

9. This is a procedure named `eval/sandbox`. It takes three arguments, the expression to evaluate, the environment name (a quoted list), and the amount of time allowed. It evaluates the expression in a copy of the given environment using `eval` and gives it a time limit to evaluate through the use of an engine. If the time expires, `eval/sandbox` throws a timeout condition.

⟨∗⟩ ≡

```
(define (eval/sandbox exp env time)
  (let ([eng (make-engine
              (lambda ()
                (parameterize ([current-eval
                                (lambda (x .  ignore) (compile x env))])
                  (eval exp))))])
    ⟨Run Timeout Engine 8⟩))
```

10. Here are basic `eval/sandbox` tests

⟨Test eval/sandbox⟩ ≡

```
(test-begin "eval/sandbox")
(let ()
  (define sdbx (sandbox '(only (chezscheme) iota)))
  (test-assert
    "eval/sandbox iota 5 with 1000000 time"
    (eval/sandbox '(iota 5) sdbx 1000000))
  (test-error
    "that eval properly errors when not enough time"
    (eval/sandbox '(iota 100) sdbx 5))
  (test-end "eval/sandbox"))
```

11. This procedure, `load/sandbox`, loads a file into a safe environment during a given amount of time. It takes three arguments: the file to load, the environment to load it in, and the given amount of time. It uses the same timeout engine defined above to `eval` each expression in the file

⟨∗⟩ ≡

```
(define (load/sandbox file env time)
  (let ([eng (make-engine
              (lambda ()
                (parameterize ([current-eval
                                (lambda (x .  ignore) (compile x env))])
                  (load file))))])
    ⟨Run Timeout Engine 8⟩))
```

12. Here are basic `load/sandbox` tests

⟨Test load/sandbox⟩ ≡

```
(test-begin "test-load/sandbox")
(let ()
  (define file "example_tests/example_a10/c211-lib.ss")
  (define sdbx (sandbox '(only (chezscheme) iota)))
  (test-assert
    "load/sandbox can open c211-lib with 1000 time"))
(test-end "test-load/sandbox")
```

13. A flagged binding is one that a student should not be using, and that we want to explicitly check for, catch, and report. We do this by making the binding a macro that always expands into an error.

⟨Define flagged⟩ ≡

```
#'(begin
    (define-syntax (fid x) #'(raise (illegal-term 'fid)))
```

```
(...  ...))
```

Captures: `fid`

14. `library/sandbox` does most of the work. It takes a list of imports, or terms we want available, and `flags`, terms we don't. It takes and loads them into a library to be used in the sandbox.

⟨∗⟩ ≡

```
(define-syntax (library/sandbox x)
  (syntax-case x (import flag)
    [(k name (import imp ...)  (flag fid ...))
     (with-implicit (k library export)
       (with-syntax ([(exp ...)  ⟨Compute export list 7⟩])
         #'(library name
             (export fid ...  exp ...)
             (import
               imp
               ...
               (only
                 (chezscheme)
                 define-syntax
                 condition
                 make-message-condition)
               (only (mags sandbox) illegal-term))
             #,⟨Define flagged 13⟩))])))
```

15. Convienience procedures. A `flag` is recognized by the sandbox as a unwanted procedure that should not be permitted in the environment.

⟨∗⟩ ≡

```
(define-syntax flag
  (lambda (x)
    (errorf #f "misplaced aux keyword ~a" (syntax->datum x))))
```

16. `sandbox` is a small procedure that simply takes a name for an environment we want to be used in the sandbox, in effect making it mutable. This should be called on any environment that will be passed to `eval/sandbox` or `load/sandbox`.

⟨∗⟩ ≡

```
(define (sandbox x) (copy-environment (environment x)))
```

17. Following is the test suite for this library.

⟨∗⟩ ≡

```
(define (run-sandbox-tests)
  (parameterize ([test-runner-current (test-runner-simple)])
    ⟨Test eval/sandbox 10⟩
    ⟨Test load/sandbox 12⟩))
```

This concludes the definition of (mags sandbox).