procedure-surface egg

Generic Programming Support
Extension for Chicken Scheme
Version 0.1

Kon Lovett

Table of Contents

1 About this egg

1.1 Version history

`0.1` Initial release

1.2 Requirements

This egg requires the following extensions:

`lookup-table`, `syntax-case`, `misc-extn`

1.3 Usage

Load this egg like so:

```
(require-extension procedure-surface)
```

2 Documentation

Warning - procedure surface is experimental!

The procedure-surface egg provides a facility for generic programming in Scheme, similar in purpose to the "structures" egg. Unlike the normal pattern of use for "structures" this facility will delay loading of any library or extension until an actual binding is required. The use of a 'define' form as a procedure reference with 'make-procedure-means' is not supported, unlike the analogous 'structure' from "structures".

Odd names are used for parsimony. Read 'interface' for 'surface', and 'implementation' for 'means'.

2.1 Syntax Interface

`define-procedure-surface` [macro]

```
(define-procedure-surface NAME PROC-SYM CONTRACT ... [KEY-ARG ...])
```

Sets the symbol `NAME` to the described procedure surface. Uses `NAME` as the keyword argument #:name for the surface. See 'make-procedure-surface' for argument descriptions.

`declare-procedure-means` [macro]

```
(declare-procedure-means NAME SURFACE PROC-SYM PROC-REF ... [KEY-ARG ...])
```

Sets the symbol `NAME` to the described procedure surface means. See 'make-procedure-means' for argument descriptions.

`call-thru-procedure-means` [macro]

```
(call-thru-procedure-means MEANS PROC-SYM [ARG ...])
```

Invoke the procedure identified by `PROC-SYM` in the `MEANS` with the `ARG ...` list.

`apply-thru-procedure-means` [macro]

```
(apply-thru-procedure-means MEANS PROC-SYM ARG ...)
```

Apply the procedure identified by `PROC-SYM` in the `MEANS` to the `ARG ...` list.

`let-procedure-means` [macro]

```
(let-procedure-means ([(PROC-SYM ...) MEANS] ...) BODY ...)
```

Bind the local symbol(s) `PROC-SYM ...` to the corresponding procedure(s) in the `MEANS` and evaluate the `BODY`.

`call/means` [macro]

```
(call/means MEANS PROC-SYM [ARG ...])
```

Abbreviation of 'call-thru-procedure-means'.

`apply/means` [macro]

```
(apply/means MEANS PROC-SYM ARG ...)
```

Abbreviation of 'apply-thru-procedure-means'.

`let/means` [macro]

```
(let/means ([(PROC-SYM ...) MEANS] ...) BODY ...)
```

Abbreviation of 'let-procedure-means'.

2.2 Direct Call Interface

`make-procedure-signature` [procedure]

```
(make-procedure-signature IDENTIFIER CONTRACT)
```

Returns a procedure signature for `IDENTIFIER` with `CONTRACT`.

The contract grammar is described below. Can be null, '(), for no contract.

`procedure-signature?` [procedure]

```
(procedure-signature? OBJECT)
```

Is `OBJECT` a procedure signature?

`procedure-signature-identifier` [procedure]

```
(procedure-signature-identifier SIGNATURE)
```

Returns the procedure signature identifier of the `SIGNATURE`.

`procedure-signature-contract` [procedure]

```
(procedure-signature-contract SIGNATURE)
```

Returns the procedure signature contract of the `SIGNATURE`.

`make-procedure-surface` [procedure]

```
(make-procedure-surface PROC-SYM CONTRACT ... [#:immutable #f] [#:name #f])
```

Creates and returns a procedure surface. A collection of procedure identifiers and contracts.

`immutable:`
: A boolean. Immutable (#t) or mutable (#f).

`name:`
: A symbol or string. The name of the surface. When missing a name of the form "ps#" will be created.

`PROC-SYM`
: A symbol. The procedure identifier.

`CONTRACT`
: A list. The source form of a procedure signature. The contract grammar is described below. null, '(), for no contract.

`procedure-surface?` [procedure]

```
(procedure-surface? OBJECT)
```

Is the `OBJECT` a procedure surface?

`procedure-surface-name` [procedure]

```
(procedure-surface-name SURFACE)
```

Returns the name, or names as list if composite, of the `SURFACE`.

`procedure-surface-immutable?` [procedure]

```
(procedure-surface-immutable? SURFACE)
```

Is the `SURFACE` immutable?

`procedure-surface-mutable?` [procedure]

```
(procedure-surface-mutable? SURFACE)
```

Is the `SURFACE` mutable?

`procedure-surface-ref` [procedure]

```
(procedure-surface-ref SURFACE PROC-SYM)
```

Returns the procedure signature of the `SURFACE`.

`procedure-surface-set!` [procedure]

```
(procedure-surface-set! SURFACE PROC-SYM CONTRACT ...)
```

Adds or updates procedure signatures in `SURFACE`.

`procedure-surface-delete!` [procedure]

```
(procedure-surface-delete! SURFACE PROC-SYM)
```

Removes the procedure signature for `PROC-SYM` from `SURFACE`.

`procedure-surface->alist` [procedure]

```
(procedure-surface->alist SURFACE)
```

Returns a alist, (<procedure-symbol> . <procedure-signature>), from `SURFACE`.

`make-composite-procedure-surface` [procedure]

```
(make-composite-procedure-surface SURFACE ...)
```

Returns a procedure surface, the combination of `SURFACE ...` set. Should any surface be immutable then the composite is immutable.

`composite-procedure-surface?` [procedure]

```
(composite-procedure-surface? SURFACE)
```

Is the `SURFACE` a composite?

`make-procedure-means` [procedure]

```
(make-procedure-means SURFACE PROC-SYM PROC-REF ... [#:immutable #f] [#:extension
```

Supply procedures for the `SURFACE`.

`immutable:`
: A boolean. Immutable (#t) or mutable (#f).

`extension:`
: A symbol or string. The name of the extension.

`library:` A symbol or string. The name of the library.

`pathname:`
: A string. The absolute pathname of the extension or library. Required when the extension or library is not in the Chicken repository.

`PROC-SYM` A symbol. The procedure identifier. Must match a procedure identifier from SURFACE.

`PROC-REF` A boolean, symbol, or procedure. The procedure alias when a symbol. When boolean use the procedure identifier as the alias. Otherwise this should be a procedure.

`procedure-means?` [procedure]

```
(procedure-means? OBJECT)
```

Is the `OBJECT` a procedure surface?

`procedure-means-immutable?` [procedure]
`(procedure-means-immutable? MEANS)`

Are the `MEANS` immutable?

`procedure-means-mutable?` [procedure]
`(procedure-means-mutable? MEANS)`

Are the `MEANS` mutable?

`procedure-means-alias` [procedure]
`(procedure-means-alias MEANS PROC-SYM)`

Returns the alias of `PROC-SYM` in `MEANS`.

`procedure-means-ref` [procedure]
`(procedure-means-ref MEANS PROC-SYM)`

Returns the current binding of `PROC-SYM` in the `MEANS`, which maybe the unbound value.

`procedure-means-closure` [procedure]
`(procedure-means-closure MEANS PROC-SYM)`

Forces a load, if necessary. Returns the current binding of `PROC-SYM` in the `MEANS`, which maybe the unbound value.

`procedure-means-implements` [procedure]
`(procedure-means-implements MEANS)`

Returns the procedure surface, or a list of procedure surface when composite, that the `MEANS` implements.

`procedure-means-complete?` [procedure]
`(procedure-means-complete? MEANS)`

Are all procedures in the procedure surface(s) of the `MEANS` declared?

`procedure-means-bound?` [procedure]
`(procedure-means-bound? MEANS)`

Are all the procedures in the `MEANS` bound?

`procedure-means-incompletes` [procedure]
`(procedure-means-incompletes MEANS)`

Returns an alist, (<procedure identifier> . <procedure surface>), of all procedures in the procedure surface(s) of the `MEANS ...` that are undeclared.

`procedure-means-unbounds` [procedure]
`(procedure-means-unbounds MEANS)`

Returns an alist, (<procedure-symbol> . <procedure-surface>), of all the unbound procedures in `MEANS`.

`procedure-means-incomplete-closure?` [procedure]
`(procedure-means-incomplete-closure? MEANS PROC-SYM)`

Is the procedure identified by `PROC-SYM` without a declaration in the `MEANS`?

`procedure-means->alist` [procedure]
`(procedure-means->alist MEANS)`

Returns an alist, (<procedure-symbol> . <procedure-alias/closure), from the `MEANS`.

`procedure-means-set!` [procedure]
`(procedure-means-set! MEANS PROC-SYM PROC-REF ...)`

Adds or updates procedures for an existing `MEANS`.

`procedure-means-delete!` [procedure]
`(procedure-means-delete! MEANS PROC-SYM)`

Removes the procedure `PROC-SYM` from the `MEANS`.

`procedure-means-load!` [procedure]
`(procedure-means-load! MEANS)`

Load of any libraries and extensions required by the `MEANS`.

`make-composite-procedure-means` [procedure]
`(make-composite-procedure-means MEANS ...)`

Returns a procedure means, the combination of `MEANS ...` set. Should any means be immutable then the composite is immutable.

`composite-procedure-means?` [procedure]
`(composite-procedure-means? MEANS)`

Are the `MEANS` a composite?

`procedure-unbound?` [procedure]
`(procedure-unbound? PROCEDURE)`

Is this surface means procedure loaded?

`procedure-identifier->closure` [procedure]
`(procedure-identifier->closure SYMBOL)`

Returns the top level binding, if any, for the procedure named `SYMBOL`.

2.3 Signature Types

The signature type system supports multiple-inheritance, via the 'extends' property.

`build-signature-type-builtins` [procedure]
`(build-signature-type-builtins)`

Creates all the builtin types.

`make-signature-type` [procedure]
`(make-signature-type NAME #!optional (EXTENDS #f) (PREDICATE #f) (SPECIALIZER #f)`

Creates a new signature type object.

`NAME` A symbol. The name of the new type.

`EXTENDS` A symbol or list. The type(s) that the new type is extending.

`PREDICATE`
: A procedure. Determines whether an arbitrary object is a value of the new type.

`SPECIALIZER`
: A procedure. Determines whether a specialization for the new type is valid.

`signature-type?` [procedure]

```
(signature-type? OBJECT)
```

Is `OBJECT` a signature type?

`signature-extended-type?` [procedure]

```
(signature-extended-type? TYPE/NAME)
```

Does a type extend `TYPE/NAME`.

`signature-leaf-type?` [procedure]

```
(signature-leaf-type? TYPE/NAME)
```

Does `TYPE/NAME` not extend a type?

`signature-type-a-kind-of?` [procedure]

```
(signature-type-a-kind-of? TYPE/NAME-SUB TYPE/NAME-SUPER)
```

Does `TYPE/NAME-SUB` extend `TYPE/NAME-SUPER`?

The most distant ancestor of all Scheme types is 'object'. However, roots are not considered when determining the kind of relationship from an intermediate type. So to check if a type is a kind of 'object' do so directly.

`signature-type-ref` [procedure]

```
(signature-type-ref NAME)
```

Returns the signature type for `NAME`.

`signature-type-name` [procedure]

```
(signature-type-name TYPE)
```

Returns the name of `TYPE`.

`signature-type-predicate` [procedure]

```
(signature-type-predicate TYPE/NAME)
```

Returns the predicate for `TYPE/NAME`.

`signature-type-specializer` [procedure]

```
(signature-type-specializer TYPE/NAME)
```

Returns the specializer for `TYPE/NAME`.

`signature-type-extends` [procedure]

```
(signature-type-extends TYPE/NAME)
```

Return a list of the types that `TYPE/NAME` extends.

`signature-type-extension` [procedure]
`(signature-type-extension TYPE/NAME)`

Return a list of the types that extend `TYPE/NAME`.

`signature-type-delete!` [procedure]
`(signature-type-delete! TYPE/NAME)`

Remove the `TYPE/NAME`.

`signature-type-replace!` [procedure]
`(signature-type-replace! TYPE/NAME #!optional (NEW-NAME #f) (EXTENDS #f) (PREDICA`

Replace the existing `TYPE/NAME` with a new definition.

`make-signature-contract` [procedure]
`(make-signature-contract CONTRACT)`

Validates the source form of a contract and returns the internal form.

2.4 Procedure Signature Grammar

`contract` <map> | (or <map> ...)

`type` <symbol> | <specialization>

`specialization`
<map> | (<type> [<parameter> ...]) | (or <type> ...)

`parameter`
<object>

`domain` [<type> ...] [#!optional <type> ...] [#!rest list | (list <type> ...)] [#!key (<keyword> <type>) ...]

`range` <type> | (values <type> ...)

`map` (-> <domain> <range> [<throw> ...])

`throw` (signals <exception> ...) | (aborts <exception> ...)

`exception`
<symbol> | (<symbol> ...)

A <map> is a specialization of '->', the type of a procedure.

'()' is a synonym of 'null' for this grammar.

Example: (-> (array (rank 2)) object (values integer complex vector object)) - a procedure (<map>) taking a matrix & any object, returning 4 multiple values.

Builtin Types

- object
- void
- structure
- boolean
- symbol

- keyword
- char
- string
- vector
- list
- pair
- null
- port
- input-port
- output-port
- eof
- procedure
- macro
- continuation
- promise
- environment
- read-table
- hash-table
- queue
- condition
- number
- exact
- inexact
- real
- integer
- fixnum
- flonum
- rational
- complex
- u8vector
- s8vector
- u16vector
- s16vector
- u32vector
- s32vector
- f32vector
- f64vector
- char-set
- mmap

- terminal-port
- tcp-listener
- thread
- lock
- mutex
- condition-variable
- time
- regexp
- pointer
- tagged-pointer
- swig-pointer
- locative
- byte-vector
- extended-procedure
- object-evicted
- record
- clos-object
- class
- method
- generic
- c++-object
- date
- array
- future

3 License

Index