4.12 Boxes

> Boxes in **The Racket Guide** introduces boxes.

A *box* is like a single-element vector, normally used as minimal mutable storage.

A literal or printed box starts with `#&`. See Reading Boxes for information on `read`ing boxes and Printing Boxes for information on `print`ing boxes.

`(box? v) → boolean?` procedure

`v : any/c`

Returns `#t` if `v` is a box, `#f` otherwise.

`(box v) → box?` procedure

`v : any/c`

Returns a new mutable box that contains `v`.

`(box-immutable v) → (and/c box? immutable?)` procedure

`v : any/c`

Returns a new immutable box that contains `v`.

`(unbox box) → any/c` procedure

`box : box?`

Returns the content of `box`.

For any `v`, `(unbox (box v))` returns `v`.

`(set-box! box v) → void?` procedure

`box : (and/c box? (not/c immutable?))`

`v : any/c`

Sets the content of `box` to `v`.

`(box-cas! box old new) → boolean?` procedure

`box : (and/c box? (not/c immutable?) (not/c impersonator?))`

`old : any/c`

```
new : any/c
```

Atomically updates the contents of `box` to `new`, provided that `box` currently contains a value that is `eq?` to `old`, and returns `#t` in that case. If `box` does not contain `old`, then the result is `#f`.

If no other threads or futures attempt to access `box`, the operation is equivalent to

```
(and (eq? old (unbox loc)) (set-box! loc new) #t)
```

When Racket is compiled with support for futures, `box-cas!` uses a hardware *compare and set* operation. Uses of `box-cas!` be performed safely in a future (i.e., allowing the future thunk to continue in parallel).