CL-CTRIE

CL-CTRIE is a common-lisp implementation of the CTrie unordered map data-structure described in the paper '*Concurrent Tries with Efficient Non-Blocking Snapshots*, (c) ACM 2-25-2012' by Prokopec, Bronson, Bagwell, and Odersky.

Overview

A brief overview of general ctrie concepts and existing implementations is available through its page on wikipedia, of which the following is a brief excerpt:

> The Ctrie data structure is a non-blocking concurrent hash array mapped trie based on single-word compare-and-swap instructions in a shared-memory system. It supports concurrent LOOKUP, INSERT and REMOVE operations. Just like the hash array mapped trie, it uses the entire 32-bit space for hash values thus having low risk of hashcode collisions... Ctries have been shown to be comparable in performance with concurrent skip lists, concurrent hash tables and similar data structures in terms of the lookup operation... However, they are far more scalable than most concurrent hash tables where the insertions are concerned. Most concurrent hash tables are bad at conserving memory - when the keys are removed from the hash table, the underlying array is not [reduced in size]. Ctries have the property that the allocated memory is always a function of only the current number of keys in the data-structure. Ctries have logarithmic complexity bounds of the basic operations... with a low constant factor due to [large dispersal ratio, (32^n arcs at level n)]. Ctries support a lock-free, linearizable, constant-time SNAPSHOT operation... This is a breakthrough in concurrent data-structure design, since other existing concurrent data-structures do not support snapshots. [This provides the means to support features such as] lock-free, linearizable iterator, size and clear operations. [This is superior to other] existing concurrent data-structures [which require the use of global locks [for exclusive, blocking semantics for update access] permitting... [no concurrent readers or writers] during any [update, insert, remove or other modifications]. In particular, Ctries have an O(1) ITERATOR creation operation, O(1) CLEAR operation, O(1) DUPLICATE operation and an amortized O(log n) operation for SIZE-RETRIEVAL.

Platform

Currently the lisp platform supported by cl-ctrie is SBCL version 1.0.55 or greater hosted on x86/x86-64 architecture. Support could easily be entended to include other common-lisp implementations that offer atomic compare-and-swap functionality, notably LispWorks 5.x/6.x, which is also well instrumented with lock-free, atomic primitives, although this is not necessarily a high priority for the initial development cycle.

Status

All unit tests should succeed, and parallelism has been tested for 1, 2, 4, and 8 threads on an 8-core system (Dual Intel(R) Xeon(R) CPU E5462 @ 2.80GHz), SBCL version 1.0.57.56-2273f3a, and Mac OS X Server version 10.6.8.

```
Starting test run on Tuesday, September 4, 2012 09:02:59 PM EDT
-----------------------------------------------------------------


CHECK-ALET-FSM: 7 assertions passed, 0 failed.
CHECK-ATOMIC-CLEAR: 6 assertions passed, 0 failed.
CHECK-ATOMIC-UPDATE: 1 assertions passed, 0 failed.
```

```
CHECK-BULK-INSERT/DROP: 1048579 assertions passed, 0 failed.
CHECK-BULK-INSERT/LOOKUP: 1048578 assertions passed, 0 failed.
CHECK-BYTE-VECTOR-HEX-STRING-ROUNDRIP: 10 assertions passed, 0 failed.
CHECK-CATCH-CASE: 128 assertions passed, 0 failed.
CHECK-CTRIE-SMOKE-TEST: 31 assertions passed, 0 failed.
CHECK-DEPTH-AND-SIMPLE-EXTENSION: 8 assertions passed, 0 failed.
CHECK-EXTENSION/RETRACTION/LNODE-CHAINING: 14 assertions passed, 0 failed.
CHECK-FBIND: 3 assertions passed, 0 failed.
CHECK-FLAG-ARC-POSITION: 165 assertions passed, 0 failed.
CHECK-FLAG-COMPUTATION: 12 assertions passed, 0 failed.
CHECK-LNODE-INSERTED/REMOVED: 218 assertions passed, 0 failed.
CHECK-LNODE-LENGTH/ENLIST: 8 assertions passed, 0 failed.
CHECK-LNODE-SEARCH: 4 assertions passed, 0 failed.
CHECK-PARALLEL-INSERT-PARALLEL-DROP: 1048588 assertions passed, 0 failed.
CHECK-PARALLEL-INSERT-PARALLEL-LOOKUP: 1048584 assertions passed, 0 failed.
CHECK-SIMPLE-INSERT/LOOKUP: 170 assertions passed, 0 failed.
CHECK-SIMPLE-INSERT/LOOKUP/DROP: 255 assertions passed, 0 failed.
CHECK-TABLE-ABSTRACTION-FIXTURES: 3 assertions passed, 0 failed.
CHECK-TIMING-COLLECTION-FIXTURES: 18 assertions passed, 0 failed.

TOTAL: 4195390 assertions passed, 0 failed, 0 execution errors.
```

Ideosyncrasies

Perhaps most ideosyncrasies of this common-lisp ctrie implementation as compared with the original, written in Scala, result from the efforts I have taken to, where feasible, adopt an approach emphasizing a more functional oriented decomposition of the algortithm, written in a manner that is more closely representative of ideomatic, common-lisp coding style. For example, rather than expose a general purpose GCAS and RDCSS api, these protocols are incorporated into ctrie-specific abstractions. For **GCAS** the exposed api includes: `INODE-READ` `INODE-MUTATE` and `INODE-COMMIT` and for **RDCSS:** `ROOT-NODE-ACCESS` `ROOT-NODE-REPLACE` and `ROOT-NODE-COMMIT.` The liberties I have taken have the intended benefit of providing an interface that is much easier to digest, understand, remember, and work with (at least for me) than direct exposure in imperative style of the intricate mechanations that underlie the ctrie algorithm. On the other hand, the further one strays from a direct translation of the original (verified) ctrie implementation, the greater the likelihood of introducing bugs into an environment (lock-free concurrent data structure development) in which bugs can be extremely subtle and notoriously difficult to detect. I have attempted to strike an appropriate balance between these conflicting concerns, and I intend to mitigate the risk, at least in part, through continued development of an extensive arsenal of regression tests and benchmarking facilities.

In addition, there are a few differences in the feature set that is provided -- such as a suite of mapping operators in leiu of a Java-style Iterator. For the most part I expect that these changes will be preferable to developers accustomed to a more 'lispy' coding style.

For additional insight into the specifics unique to this ctrie implementation, an abbreviated reference to a number of internal details may be found in the section Internal Reference of this document, or, of course, by referring to the comprehensive documentation that is provided as part of this distribution.

Source Files

The following outline provides a general description of the constituent source files of the cl-ctrie repository and their respective purpose.

Required

The following files are the core sources currently necessary for correct ctrie operation.

- **`ctrie-package.lisp:`** Package Definition
- **`ctrie.lisp:`** Structural Implementation
- **`ctrie-util.lisp:`** Supporting Utilities

Supplemental

The following files define extended functionality, management, and analysis facilities supporting the development of CL-CTRIE.

- `ctrie-cas.lisp:` SBCL CAS extensions
- `ctrie-lambda.lisp:` Experiments with Stateful Protocols
- `ctrie-doc.lisp:` Automated Documentation Support
- `ctrie-test.lisp:` Test and Performance Measurement

Performance

To date, the principal concern has been to fully achieve a working and robust implementation of the CTRIE algorithm, with only the occaisional profiling run from time to time in order to at least get a general sense of where the hot-spots are and peace of mind that there are at least no glaring bottlenecks that seem out of hand. A basic profiling exercise of the following activities is reported below.

```
1. construct a new ctrie instance
2. perform one-million insertions of (fixnum) key/value pairs
3. perform one-million lookups verifying retrieval of every value
4. perform one-million removals verifying proper contraction of the structure
5. verify the resulting ctrie is identical to when initially created.
```

```
  seconds  |     gc     |     consed    |    calls    |  sec/call  |  name
-------------------------------------------------------------
     3.810 |      0.000 |           384 | 12,000,000 |   0.000000 | CTHASH
     1.770 |      0.000 |         1,040 | 12,000,000 |   0.000000 | FLAG
     1.332 |      0.925 |   261,160,000 |  1,000,000 |   0.000001 | CTRIE-PUT
     0.955 |      0.000 |             0 |  5,101,473 |   0.000000 | FIND-CTRIE-ROOT
     0.654 |      0.071 |   357,490,000 |  2,101,473 |   0.000000 | MAKE-CNODE
     0.600 |      0.000 |            48 |  2,067,648 |   0.000000 | INODE-COMMIT
     0.543 |      0.040 |    85,981,184 |  2,101,473 |   0.000000 | %MAKE-CNODE
     0.521 |      0.000 |           176 |  2,033,824 |   0.000000 | CTEQUAL
     0.494 |      0.000 |             0 | 12,000,000 |   0.000000 | FLAG-ARC-POSI-
TION
     0.400 |      0.237 |    20,388,048 |  1,000,000 |   0.000000 | MAKE-SNODE
     0.369 |      0.054 |   143,871,728 |  2,101,473 |   0.000000 | MAKE-REF
     0.350 |      0.000 |             0 | 11,966,176 |   0.000000 | FLAG-PRESENT-P
     0.319 |      0.000 |             48 |  2,000,000 |   0.000000 | LEAF-NODE-KEY
     0.197 |      0.000 |            112 |  1,000,000 |   0.000000 | SNODE
     0.172 |      0.000 |              0 |  2,000,000 |   0.000000 | LEAF-NODE-VALUE
     0.147 |      0.000 |             32 |  2,101,473 |   0.000000 | CTSTAMP
     0.066 |      0.000 |              0 |  2,101,473 |   0.000000 | FLAG-VECTOR
     0.032 |      0.000 |            112 |     33,825 |   0.000001 | MAKE-INODE
     0.015 |      0.000 |            208 |     33,824 |   0.000000 | ENTOMB
     0.008 |      0.000 |              0 |     33,824 |   0.000000 | RESURRECT
     0.000 |      0.000 |         65,504 |     33,824 |   0.000000 | MAKE-TNODE
     0.000 |      0.000 |              0 |          2 |   0.000000 | MAP-NODE
     0.000 |      0.000 |     42,082,064 |  1,000,000 |   0.000000 | CNODE-EXTENDED
     0.000 |      0.000 |             64 |     67,648 |   0.000000 | CNODE-UPDATED
     0.000 |      0.000 |            256 |  1,000,000 |   0.000000 | CNODE-TRUNCATED
     0.000 |      0.000 |              0 |     33,825 |   0.000000 | %MAKE-INODE
     0.000 |      0.000 |              0 |          1 |   0.000000 | %MAKE-CTRIE
     0.000 |      0.000 |              0 |     33,824 |   0.000000 | CLEAN-PARENT
     0.000 |      0.000 |          1,536 |  4,000,000 |   0.000000 | %LOOKUP
     0.000 |      0.000 |          1,376 |  3,966,176 |   0.000000 | %REMOVE
     0.000 |      0.000 |              0 |          1 |   0.000000 | MAKE-CTRIE
     0.000 |      0.000 |              0 |          1 |   0.000000 | CTRIE-MAP-KEYS
     0.000 |      0.000 |              0 |          1 |   0.000000 | CTRIE-MAP
     0.000 |      0.232 |    320,038,576 |  1,000,000 |   0.000000 | CTRIE-GET
     0.000 |      0.000 |              0 |          1 |   0.000000 | CTRIE-SIZE
     0.000 |      0.601 |    229,090,464 |  1,000,000 |   0.000000 | CTRIE-DROP
-----------------------------------------------------------------
    12.750 |      2.160 |  1,460,172,960 | 86,913,263 |            | Total
```

Documentation

Comprehensive, HTML based documentation may be found at the project relative pathname `doc/api/index.html`. In order to enable support for supplemental documentation related features, such as the ability to dynamically regenerate the provided documentation files incorporating all updates and changes as may be present in the source code, a (lightly) enhanced distribution of CLDOC is required and may be obtained on github.

The following sections provide a compact overview of the user api and a reference to some internal definitions of interest.

User API Reference

The user api of cl-ctrie should be largely familiar to the average common-lisp programmer. Nearly all exported symbols of the CL-CTRIE package begin with the prefix "ctrie" and thus can be convenientely incorporated via USE-PACKAGE or equivalent package definition. The following definitions comprise a quick reference to the the public user api:

[structure] `CTRIE ()`

A CTRIE structure is the root container that uniquely identifies a CTRIE instance, and contains the following perameters which specify the definable aspects of each CTRIE:

- `READONLY-P` if not `NIL` prohibits any future modification or cloning of this instance.
    - `TEST` is a designator for an equality predicate that will be applied to disambiguate and determine the equality of any two keys. It is recommened that this value be a symbol that is fboundp, to retain capability of externalization (save/restore). At present, though, this is not enforced and a function object or lambda expression will also be accepted, albeit without the ability of save/restore.
    - `HASH` is a designator for a hash function, which may be desirable to customize when one has specific knowledge about the set of keys which will populate the table. At this time, a 32-bit hash is recommended as this is what has been used for development and testing and has been shown to provide good performance in practice. As with `TEST` it is recommended that `HASH` be specified by a symbol that is fboundp.
    - `ROOT` is the slot used internally for storage of the root inode structure that maintains the reference to the contents of the ctrie proper. The ctrie-root must only be accessed using the *RDCSS ROOT NODE PROTOCOL* defined by the top-level entry-points `ROOT-NODE-ACCESS` and `ROOT-NODE-REPLACE`

[function] `MAKE-CTRIE (&REST ARGS &KEY NAME ROOT (READONLY-P NIL) (TEST 'EQUAL) (HASH 'SXHASH) &ALLOW-OTHER-KEYS)`

CREATE a new CTRIE instance. This is the entry-point constructor intended for use by the end-user.

[function] `CTRIE-P (OBJECT)`

Returns T if the specified object is of type ctrie.

[function] `CTRIE-TEST (CTRIE)`

Returns the test of the specified ctrie

[function] `CTRIE-HASH (CTRIE)`

Returns the hash of the specified ctrie

[function] `CTRIE-READONLY-P (CTRIE)`

Returns and (with setf) changes the readonly-p of the specified ctrie

[function] `CTRIE-PUT (CTRIE KEY VALUE)`

Insert a new entry into CTRIE mapping KEY to VALUE. If an entry with key equal to KEY aleady exists in CTRIE, according to the equality predicate defined by `CTRIE-TEST` then the priorbmapping will be replaced by VALUE. Returns `VALUE` representing the mapping in the resulting CTRIE

[function] `CTRIE-GET (CTRIE KEY)`

[function] `CTRIE-DROP (CTRIE KEY)`

Remove KEY and it's value from the CTRIE.

[macro] `CTRIE-DO ((KEY VALUE CTRIE &KEY ATOMIC) &BODY BODY)`

> Iterate over (key . value) in ctrie in the manner of dolist. ;;; EXAMPLE: (ctrie-do (k v ctrie) ;;; (format t "~&~8S => ~10S~%" k v))

[function] `CTRIE-MAP (CTRIE FN &KEY ATOMIC &AUX ACCUM)`

[function] `CTRIE-MAP-KEYS (CTRIE FN &KEY ATOMIC)`

[function] `CTRIE-MAP-VALUES (CTRIE FN &KEY ATOMIC)`

[function] `CTRIE-KEYS (CTRIE &KEY ATOMIC)`

[function] `CTRIE-VALUES (CTRIE &KEY ATOMIC)`

[function] `CTRIE-SIZE (CTRIE &AUX (ACCUM 0))`

[function] `CTRIE-CLEAR (CTRIE)`

[function] `CTRIE-PPRINT (CTRIE &OPTIONAL (STREAM T))`

[function] `CTRIE-TO-ALIST (CTRIE &KEY ATOMIC)`

[function] `CTRIE-TO-HASHTABLE (CTRIE &KEY ATOMIC)`

[function] `CTRIE-FROM-HASHTABLE (HASHTABLE &KEY CTRIE)`

> create a new ctrie containing the same (k . v) pairs and equivalent test function as HASHTABLE

[function] `CTRIE-FROM-ALIST (ALIST &KEY CTRIE)`

[function] `CTRIE-EMPTY-P (CTRIE)`

[function] `CTRIE-SNAPSHOT (CTRIE &KEY READ-ONLY)`

[macro] `DEFINE-CTRIE (NAME CTRIE &REST ARGS &KEY (OBJECT T) SPEC)`

> Define a 'functional' **CTRIE-LAMBDA** that combines all the the capabilities of the raw data structure with behavior and semantics one would expect of any other ordinary common-lisp function. The resulting symbol defined as 'name will be bound in three distinct namespaces: the `SYMBOL-VALUE` will be bound to the LAMBDA CLOSURE object, `SYMBOL-FUNCTION` (fdefinition) will be FBOUND to the compiled function, and the corresponding '(SETF NAME) form will be SETF-BOUND. the syntax for invoking NAME is as in a LISP1; i.e., no 'funcall' is required (but still works if you prefer). Calling `(NAME key)` returns the value mapped to key, or `NIL` just as if by `(CTRIE-GET ctrie-name key)`. Analogously when used as a setf-able place such as by `(setf (NAME key) value)` it has the equivalent behavior to the operation `(CTRIE-PUT ctrie-name key value)`. Use of this type of binding technique has some really convenient effects that I've quickly started to become quite fond of. One such idiom, for example, `(mapcar MY-CTRIE '(key1 key2 key3 key4 ...))` returns a list containing all the mapped values corresponding to the respective keys. One additional feature that I've found extremely useful is included *under the hood:* Invoking MY-CTRIE on an object of type FUNCTION will not search the ctrie for an entry having that func-

tion ast its key, but will instead APPLY that function to the actual CTRIE structure wrapped within the closure. Thus, `(MY-CTRIE #'identity)` will return the underlying ctrie as just an ordinary instance of a CTRIE STRUCTURE.
There are many other functions this is handy with, like `(MY-CTRIE #'ctrie-size)` `(MY-CTRIE #'ctrie-to-hashtable)` etc. Some additional examples are provided below.

```
  ;;;  (define-ctrie my-ctrie)
  ;;;    =>  MY-CTRIE
  ;;;
  ;;;  (describe 'my-ctrie)
  ;;;
  ;;;     CL-CTRIE::MY-CTRIE
  ;;;       [symbol]
  ;;;
  ;;;     MY-CTRIE names a special variable:
  ;;;       Value: #<CLOSURE (LAMBDA # :IN MAKE-CTRIE-LAMBDA)
{100F73261B}>
  ;;;
  ;;;     MY-CTRIE names a compiled function:
  ;;;       Lambda-list: (&REST ARGS1)
  ;;;       Derived type: FUNCTION
  ;;;
  ;;;     (SETF MY-CTRIE) names a compiled function:
  ;;;       Lambda-list: (VALUE KEY)
  ;;;       Derived type: (FUNCTION (T T) *)
  ;;;
  ;;;
  ;;;   (my-ctrie :HONG-KONG :FOOY)
  ;;;     =>  :FOOY
  ;;;
  ;;;   (my-ctrie :HONG-KONG)
  ;;;     =>  :FOOY ; T
  ;;;
  ;;;   (map 'list #'eval (mapcar #`(my-ctrie ,a1 ,a1) (iota 12)))
  ;;;     =>  (0 1 2 3 4 5 6 7 8 9 10 11)
  ;;;
  ;;;   (mapcar my-ctrie (iota 12))
  ;;;     =>  (0 1 2 3 4 5 6 7 8 9 10 11)
```

[function] `CTRIE-LAMBDA-SPAWN (SELF &KEY READ-ONLY)`

Causes the atomic clone of enclosed ctrie structure and builds a new lexical closure to operate on it. Does not bother to reproduce fancy (expensive) object, class, bindings, but provides almost identical functionality. May be used to more efficintly distribute workload in parallel

[macro] `CTRIE-LAMBDA (&ONCE CTRIE &REST REST)`

Pandoric Object and Inter-Lexical Communication Protocol this macro builds the most unencumbered and widely applicable 'purist edition' Of our PLAMBDA based form. Even as such, a lot of care has been given to many subtle ways it has been refined to offer the most convenient and natural tool possible.

```
     ;;; (plambda (#<CLOSURE (LAMBDA (&REST ARGS)) {100929EB1B}> )
     ;;;
     ;;; DISPATCHING to FUNCTIONAL MAPPING:
     ;;;   (IF (REST ARGS)
     ;;;          (APPLY ARG (REST ARGS))
     ;;;          (FUNCALL ARG #'IDENTITY)) =>
     ;;; ------------------------------------------------------------
     ;;; INITIALIZING PLAMBDA
     ;;; ------------------------------------------------------------
     ;;;   IT => #S(CTRIE
     ;;;                :READONLY-P NIL
     ;;;                :TEST EQUAL
     ;;;                :HASH SXHASH
     ;;;                :STAMP #<CLOSURE (LAMBDA # :IN CONSTANTLY)
{10092B516B}>
     ;;;                :ROOT #S(INODE
     ;;;                         :GEN #:|ctrie2196|
     ;;;                         :REF #S(REF
     ;;;                                 :STAMP @2012-08-
19T13:34:58.314457-04:00
     ;;;                                 :VALUE #S(CNODE :BITMAP 0 :ARCS #
())
     ;;;                                 :PREV NIL)))
     ;;;   PLIST => (:CONTAINER #<CLOSURE (LAMBDA #) {100929EACB}>
     ;;;              :TIMESTAMP @2012-08-19T13:34:58.314464-04:00)
     ;;;   STACK => (#<CLOSURE (LAMBDA #) {100929EACB}>)
     ;;;
     ;;; ------------------------------------------------------------
     ;;;  #<CLOSURE (LAMBDA (&REST #:ARGS55)) {100929EACB}>
     ;;;```
```

[generic-function] `CTRIE-LAMBDA-DISPATCH (CTRIE-LAMBDA-OBJECT)`

Returns and (with setf) changes the dispatch of the specified ctrie-lambda-object

[special-variable] `+SIMPLE-DISPATCH+ ((DLAMBDA (:FROM (ARG) ARG) (:TO (ARG) ARG) (:DOMAIN (ARG) ARG) (:RANGE (ARG) ARG)))`

[macro] `CTRIE-ERROR (CONDITION &REST ARGS)`

Signal a CTRIE related condition.

[condition] `CTRIE-ERROR (ERROR)`

Abstract superclass of CTRIE related conditions.

[condition] `CTRIE-STRUCTURAL-ERROR (CTRIE-ERROR)`

Condition designating that the CTRIE data structure has been determined to be invalid.

[condition] `CTRIE-OPERATIONAL-ERROR (CTRIE-ERROR)`

Condition for when an operational failure or inconsistency has occurred.

[condition] `CTRIE-OPERATION-RETRIES-EXCEEDED (CTRIE-OPERATIONAL-ERROR)`

Condition indicating an operation has failed the maximum number of times specified by the special-variable *retries*

[condition] `CTRIE-NOT-IMPLEMENTED (CTRIE-ERROR)`

Condition designating functionality for which the implementation has not been written, but has not been deliberately excluded.

[condition] `CTRIE-NOT-SUPPORTED (CTRIE-ERROR)`

Condition designating functionality that is deliberately not supported.

[condition] `CTRIE-INVALID-DYNAMIC-CONTEXT (CTRIE-OPERATIONAL-ERROR)`

Condition indicating an operation was attempted outside the dynamic extent of a valid enclosing WITH-CTRIE form

[condition] `CTRIE-GENERATIONAL-MISMATCH (CTRIE-STRUCTURAL-ERROR)`

Condition indicating an operation encountered an outdated or inconsistent node during its attempted traversal

[function] `CTRIE-MODIFICATION-FAILED (REASON &KEY OP PLACE)`

Signal a modification failure with the appropriate attendant metadata.

[condition] `CTRIE-MODIFICATION-FAILED (CTRIE-OPERATIONAL-ERROR)`

This condition indicates an unhandled failure of an attempt to perform stateful modification to CTRIE. The most common case in which this might occur is when such an attempt is mode on a CTRIE designated as READONLY-P. In any case, this condition represents an exception from which processing cannot continue and requires interactive user intervention in order to recover.

Internal Reference (abridged)

The following reference describes some selected internal implementation details of interest. Under normal circumstances it should not be necessary to interact with these unexported symbols unless developing an extension to cl-ctrie, but are presented here for the sake of convenience, in order to provide better insight into the ctrie structure in general, and to help illuminate significant aspects of this implementation in particular. As mentioned above, comprehensive documentation of all symbols is also provided, and should be considered the authoratative reference to the CL-CTRIE implementation.

[special-variable] `*CTRIE* (NIL)`

Within the dynamic extent of a CTRIE operation this variable will be bound to the root-container CTRIE operand. It is an error if an operation is defined that attempts access to a CTRIE without this binding, which is properly established by wrapping the operation in an appropriate WITH-CTRIE form.

[special-variable] `*RETRIES* (16)`

Establishes the number of restarts permitted to a CTRIE operation established by a WITH-CTRIE form before a condition of type CTRIE-OPERATION-RETRIES-EXCEEDED will be signaled, aborting the operatation, and requiring operator intervention to resume processing.

[special-variable] `*TIMEOUT* (2)`

Establishes the duration (in seconds) allotted to a CTRIE operation established by a WITH-CTRIE form before a condition of type CTRIE-OPERATION-TIMEOUT-EXCEEDED will be signaled, aborting the operatation, and requiring operator intervention to resume processing.

[macro] `MULTI-CATCH (TAG-LIST &BODY FORMS)`

Macro allowing catch of multiple tags at once and finding out which tag was thrown. * RETURNS: (values RESULT TAG) - RESULT is either the result of evaluationg FORMS or the value thrown by the throw form. - TAG is NIl if evaluation of the FORMS completed normally or the tag thrown and cought. * EXAMPLE: ;;; (multiple-value-bind (result tag) ;;; (multi-catch (:a :b) ;;; ...FORMS...) ;;; (case tag ;;; (:a ...) ;;; (:b ...) ;;; (t ...)))

[macro] `CATCH-CASE (FORM &REST CASES)`

User api encapsulating the MULTI-CATCH control-structure in a syntactic format that is identical to that of the familiar CASE statement, with the addition that within the scope of each CASE clause, a lexical binding is established between the symbol IT and the value caught from the throw form.

[structure] `CTRIE ()`

A CTRIE structure is the root container that uniquely identifies a CTRIE instance, and contains the following perameters which specify the definable aspects of each CTRIE:

- `READONLY-P` if not `NIL` prohibits any future modification or cloning of this instance.
    - `TEST` is a designator for an equality predicate that will be applied to disambiguate and determine the equality of any two keys. It is recommened that this value be a symbol that is fboundp, to retain capability of externalization (save/restore). At present, though, this is not enforced and a function object or lambda expression will also be accepted, albeit without the ability of save/restore.
    - `HASH` is a designator for a hash function, which may be desirable to customize when one has specific knowledge about the set of keys which will populate the table. At this time, a 32-bit hash is recommended as this is what has been used for development and testing and has been shown to provide good performance in practice. As with `TEST` it is recommended that `HASH` be specified by a symbol that is fboundp.
    - `ROOT` is the slot used internally for storage of the root inode structure that maintains the

reference to the contents of the ctrie proper. The ctrie-root must only be accessed using the *RDCSS ROOT NODE PROTOCOL* defined by the top-level entry-points `ROOT-NODE-ACCESS` and `ROOT-NODE-REPLACE`

[function] `CTRIE-P (OBJECT)`

Returns T if the specified object is of type ctrie.

[function] `CTRIE-HASH (CTRIE)`

Returns the hash of the specified ctrie

[function] `CTRIE-TEST (CTRIE)`

Returns the test of the specified ctrie

[function] `CTRIE-READONLY-P (CTRIE)`

Returns and (with setf) changes the readonly-p of the specified ctrie

[function] `CTHASH (KEY)`

Compute the hash value of KEY using the hash function defined by the CTRIE designated by the innermost enclosing WITH-CTRIE form.

[function] `CTEQUAL (X Y)`

Test the equality of X and Y using the equality predicate defined by the CTRIE designated by the innermost enclosing WITH-CTRIE form.

[macro] `WITH-CTRIE (&ONCE CTRIE &BODY BODY)`

Configure the dynamic environment with the appropriate condition handlers, control fixtures, and instrumentation necessary to execute the operations in BODY on the specified CTRIE. Unless specifically documented, the particular configuration of this dynamic environment should be considered an implementation detail and not relied upon. A particular exception, however, is that within the dynamic extent of a WITH-CTRIE form, the code implementing a CTRIE operation may expect that the special variable `*CTRIE*` will be bound to the root container of subject CTRIE. See also the documentation for `*CTRIE*`

[function] `FLAG (KEY LEVEL)`

For a given depth, LEVEL, within a CTRIE, extract the correspondant sequence of bits from the computed hash of KEY that indicate the logical index of the arc on the path to which that key may be found. Note that the logical index of the arc is most likely not the same as the physical index where it is actually located -- for that see `FLAG-ARC-POSITION`

[function] `FLAG-PRESENT-P (FLAG BITMAP)`

Tests the (fixnum) BITMAP representing the logical index of all arcs present in a CNODE for the presence of a particular arc whose logical index is represented by FLAG.

[function] `FLAG-ARC-POSITION (FLAG BITMAP)`

Given FLAG representing the logical index of an arc, and BITMAP representing all arcs present, compute a physical index for FLAG in such a manner as to always ensure all arcs map uniquely and contiguously to the smallest vector that can contain the given arcs.

[function] `FLAG-VECTOR (&OPTIONAL (CONTENT 0))`

FLAG-VECTOR is a bit-vector representation of the (fixnum) BITMAP. It is currently not used for any calculation, however it is included within each CNODE as a convenience because it makes it immediately clear from visual inspection which logical arc indexes are represented in the node. For example, from the bit-vector `#*10010000000000000000000000000000` one can easily see that the first and fourth positions are occupied, and the rest empty.

[structure] `REF ()`

Atomically Stamped Reference structure *[Herlithy, TAOMP]* that encapsulates the mutable slots within an inode. Any specific `REF` structure is, itself, never mutated. Using the `REF` structure as basis of inode implementation provides the capability to 'bundle' additional metadata in an inode while still providing atomic compare and swap using a single comparison of the aggregate `REF` instance.

- `STAMP` defines a slot containing implementation-specific metadata that may be maintained internally as a means of tracking inode modification and update behavior. It should not be referenced by user code, and the format of its contents should not be relied apon.
- `VALUE` defines a slot that contains a reference to the MAIN-NODE that the enclosing inode should be interpreted as 'pointing to'
- `PREV` defines a slot which, during the `INODE-COMMIT` phase of the *GCAS INODE PROTOCOL* maintains a reference to the last valid inode state, which may be restored, if necessary, during the course of the `INODE-READ` / `INODE-COMMIT` arbitration process

[function] `REF-P (OBJECT)`

Returns T if the specified object is of type ref.

[function] `REF-STAMP (REF)`

Returns the stamp of the specified ref

[function] `REF-VALUE (REF)`

Returns the value of the specified ref

[function] `REF-PREV (REF)`

Returns and (with setf) changes the prev of the specified ref

[structure] `FAILED-REF (REF)`

A `FAILED-REF` is a structure that is used to preserve the linkage to prior inode state following a failed GCAS. Any inode access that detects a `FAILED-REF` will immediately invoke a commit to restore the inode to the state recorded in `FAILED-REF-PREV` *[function]* `FAILED-REF-P (OBJECT)`

Returns T if the specified object is of type failed-ref.

[type] `LEAF-NODE NIL`

A LEAF-NODE represents a terminal value in a CTRIE arc. LEAF-NODEs always contain a unit-payload of CTRIE data storage; For example, an SNODE contains a key/value pair.

[type] `BRANCH-NODE NIL`

a BRANCH-NODE represents a single arc typically contained within A CNODE.

[type] `MAIN-NODE NIL`

A MAIN-NODE is a node that typically represents a specific depth or level within the ctrie and is referenced by its own unique inode.

[structure] `INODE ()`

An INODE, or 'Indirection Node' is the mutable structure representing the link between other 'main-node' structures found in a ctrie. An inode is the only type of node that may change the value of its content during its lifetime. In this implementation, all such values as may change are encapsulated within a `REF` substructure. Each inode also contains a generational descriptor object, comparible by identity only, which is used to identify the state of synchronization between the inode and the current 'generation' indicated by the root inode at the base of the CTRIE. As an inode may not change its 'gen identity' during its lifetime, this disparity with the generation of the root node will immediately result in the replacement of the inode with a new one properly synchronized with the root's `GEN` object. In this implementation, `GEN` objects are implemented by GENSYMS -- unique, uninterned symbols which inherently provide very similar symantics to those required of the generational descriptor.

- `GEN` defines a slot containing a generational descriptor object
- `REF` defines a slot containing a `REF` struct that encapsulates the mutable content within an INODE

[function] `INODE-P (OBJECT)`

Returns T if the specified object is of type inode.

[function] `INODE-GEN (INODE)`

Returns the gen of the specified inode

[function] `INODE-REF (INODE)`

Returns and (with setf) changes the ref of the specified inode

[function] `MAKE-INODE (LINK-TO &OPTIONAL GEN STAMP PREV)`

Construct a new INODE that represents a reference to the value provided by argument LINK-TO, optionally augmented with a specified generational descriptor, timestamp, and/or previous state *[macro]* `GCAS-COMPARE-AND-SET (OBJ EXPECTED NEW EXPECTED-STAMP NEW-STAMP PREV)`

A thin, macro layer abstraction over the basic compare-and-swap primitive which provides a consistent interface to the underlying inode structure and manages additional metadata, providing reasonable defaults when they are not specified.

[function] `INODE-READ (INODE)`

INODE-READ provides the top-level interface to the inode *GCAS ACCESS* api, which is the mechanism which must be used to gain access to the content of any NON-ROOT inode. For access to the root inode, refer to the RDCSS inode api `ROOT-NODE-ACCESS`. Returns as four values, the MAIN-NODE, the STAMP, the PREVIOUS STATE (if any), and the REF structure encapsulated by the inode.

[function] `INODE-MUTATE (INODE OLD-VALUE NEW-VALUE)`

INODE-MUTATE provides the top-level interface to the inode *GCAS MODIFICATION* api, which is the mechanism which must be used to effect any change in a NON-ROOT inode. For modification of the root-inode, refer to the `ROOT-NODE-REPLACE` *RDCSS ROOT NODE PROTOCOL* Returns a boolean value which indicates the success or failure of the modification attempt.

[function] `INODE-COMMIT (INODE REF)`

INODE-COMMIT implements the *GCAS COMMIT* protocol which is invoked as necessary by the `INODE-READ` and `INODE-MUTATE` entry-points. It is not meant to be invoked directly, as this would most likely result in corruption. Returns the `REF` structure representing the content of whatever root inode wound up successfully committed -- either the one requested, or one represented by a previous valid state. In order to coexist with the *RDCSS ROOT NODE PROTOCOL* this GCAS COMMIT implementation is augmented with RDCSS ABORTABLE READ semantics by a forward reference to a RDCSS-aware `ROOT-NODE-ACCESS` in order to safely compare INODE's generational descriptor with the one found in the root inode of the subject CTRIE.

[function] `SNODE (KEY VALUE)`

Construct a new SNODE which represents the mapping from domain-element KEY to range-element VALUE.

[structure] `SNODE ()`

SNODE, i.e., 'Storage Node', is the LEAF-NODE structure ultimately used for the storage of each key/value pair contained in the CTRIE. An SNODE is considered to be immutable during its lifetime.

- `KEY` defines the slot containing an element of the map's domain.
- `VALUE` defines the slot containing the range-element mapped to `KEY`

[function] `SNODE-P (OBJECT)`

Returns T if the specified object is of type snode.

[function] `SNODE-KEY (SNODE)`

Returns the key of the specified snode

[function] `SNODE-VALUE (SNODE)`

Returns the value of the specified snode

[structure] `LNODE ()`

LNODE, i.e., 'List Node', is a special structure used to enclose SNODES in a singly-linked chain when the hash-codes of the respective SNODE-KEYS collide, but those keys are determined to be unique by the `CTRIE-TEST` function defined for that ctrie. An LNODE (and therefore a chain of LNODEs) is considered to be immutable during its lifetime. The order of the list is implemented (arbitrarily) as most recently added first, analogous to `CL:PUSH`

- `ELT` defines the slot containing an enclosed SNODE
- `NEXT` defines a slot referencing the next LNODE in the chain, or `NIL` if no further LNODES remain.

[function] `LNODE-P (OBJECT)`

Returns T if the specified object is of type lnode.

[function] `LNODE-ELT (LNODE)`

Returns the elt of the specified lnode

[function] `LNODE-NEXT (LNODE)`

Returns the next of the specified lnode

[function] `ENLIST (&REST REST)`

Construct a chain of LNODE structures enclosing the values supplied. It is assumed (elsewhere) that each of these values is a valid SNODE structure.

[function] `LNODE-REMOVED (ORIG-LNODE KEY TEST)`

Construct a chain of LNODE structures identical to the chain starting with ORIG-LNODE, but with any LNODE containing an SNODE equal to KEY removed. Equality is tested as by the predicate function passed as the argument TEST. The order of nodes in the resulting list will remain unchanged.

[function] `LNODE-INSERTED (ORIG-LNODE KEY VALUE TEST)`

Construct a chain of LNODE structures identical to the chain starting with ORIG-LNODE, but ensured to contain an LNODE enclosing an SNODE mapping KEY to VALUE. If the given KEY equal to a key already present somewhere in the chain (as compared with equality predicate TEST) it will be replaced. Otherwise a new LNODE will be added. In either case, the LNODE containing `(SNODE KEY VAlUE)` will be the first node in the resulting list

[function] `LNODE-SEARCH (LNODE KEY TEST)`

Within the list of lnodes beginning with LNODE, return the range value mapped by the first SNODE containing a key equal to KEY as determined by equality predicate TEST, or `NIL` if no such key is found. As a second value, in order to support storage of `NIL` as a key, return `T` to indicate that the KEY was indeed found during search, or `NIL` to indicate that no such key was present in the list

[function] `LNODE-LENGTH (LNODE)`

Return the number of LNODES present in the chain beginning at LNODE

[structure] `TNODE ()`

A TNODE, or 'Tomb Node', is a special node structure used to preserve ordering during `CTRIE-DROP` (`%remove`) operations. Any time a TNODE is encountered during the course of a `CTRIE-GET` (`%lookup`) operation, the operative thread is required to invoke a `CLEAN` operation on the TNODE it has encountered and throw to `:RESTART` its lookup activity over again. A TNODE is considered to be immutable and may not change its value during its lifetime.

- `CELL` defines a slot which contains the entombed node structure. Only LNODE and SNODE type nodes are ever entombed

[function] `TNODE-P (OBJECT)`

Returns T if the specified object is of type tnode.

[function] `TNODE-CELL (TNODE)`

Returns the cell of the specified tnode

[structure] `CNODE ()`

A CNODE, or 'Ctrie Node' is a MAIN-NODE containing a vector of up to `2^W` 'arcs' -- i.e., references to either an SNODE or INODE structure, collectively referred to as `BRANCH-NODES.` Each CNODE also contains a (fixnum) bitmap that describes the layout of which logical indices within the total capacity of `2^W` arcs are actually allocated within that node with BRANCH-NODES physically present. For more specific details on these BITMAP and ARC-VECTOR constituents, refer to the following related functions: `FLAG` `FLAG-PRESENT-P` `FLAG-VECTOR` and `FLAG-ARC-POSITION. The CNODE structure is considered to be immutable and arcs may not be added or removed during its lifetime. The storage allocated within a CNODE is fixed and specified at the time of its creation based on the value of BITMAP during initialization

- `BITMAP`
- `ARCS`

[function] `CNODE-P (OBJECT)`

Returns T if the specified object is of type cnode.

[function] `MAKE-CNODE (&OPTIONAL (BITMAP 0))`

Construct a CNODE with internal storage allocated for the number of arcs equal to the Hamming-Weight of the supplied BITMAP parameter. If no BITMAP is provided, the CNODE created will be empty -- a state which is only valid for the level 0 node referenced by the root of the CTRIE. This constructor is otherwise never called directly, but is invoked during the course of higher-level operations such as `CNODE-EXTENDED` `CNODE-UPDATED` `CNODE-TRUNCATED` and `MAP-CNODE`

[function] `CNODE-EXTENDED (CNODE FLAG POSITION NEW-ARC)`

Construct a new cnode structure that is exactly like CNODE, but additionally contains the BRANCH-NODE specified by parameter NEW-ARC and logical index FLAG at the physical index POSITION within its vector of allocated arcs. The BITMAP of this new CNODE will be calculated as an adjustment of the prior CNODE's BITMAP to reflect the presence of this additional arc. In addition, the physical index within the extended storage vector for the other arcs present may also change with respect to where they were located in the prior CNODE. In other words, the physical index of a given arc within the compressed CNODE storage vector should never be relied upon directly, it should always be accessed by calculation based on its LOGICAL index and the current CNODE BITMAP as described in more detail by the documentation for the functions `FLAG` `FLAG-VECTOR` and `FLAG-ARC-POSITION`

[function] `CNODE-UPDATED (CNODE POSITION REPLACEMENT-ARC)`

Construct a new cnode structure identical to CNODE, but having the BRANCH-NODE physically located at POSITION within the storage vector replaced by the one specified by REPLACEMENT-ARC. Unlike `CNODE-EXTENDED` and `CNODE-TRUNCATED` the allocated storage and existing BITMAP of this CNODE will remain unchanged (as this is simply a one-for-one replacement) and correspondingly, no reordering of other nodes within the storage vector will occur

[function] `CNODE-TRUNCATED (CNODE FLAG POS)`

Construct a new cnode structure that is exactly like CNODE, but with the arc at logical index FLAG and physical storage vector location POS removed. The new CNODE will have an updated bitmap value that is adusted to reflect the removal of this arc, and the position of other arcs within the storage vector of the new CNODE will be adjusted in a manner analogous to that of `CNODE-EXTENDED` More details on this process may be found by referring to the documentation for the functions `FLAG` `FLAG-VECTOR` and `FLAG-ARC-POSITION`

[function] `MAP-CNODE (FN CNODE)`

Construct a new cnode structure that is exactly like CNODE, but with each arc (BRANCH-NODE) present in CNODE replaced by the result of applying FN to that arc. I.e., a simple functional mapping from the old CNODE by FN. As with `CNODE-UPDATED` the allocated storage and BITMAP of the resulting CNODE will remain unchanged from the original, and no physical reordering of nodes within the storage vector will occur

[function] `CNODE-CONTRACTED (CNODE LEVEL)`

The *CONTRACTION* of a CNODE is an ajustment performed when a CNODE at depth other than level 0 contains only a single SNODE arc. In such a case, that SNODE is entombed in a new TNODE, which is returned as the result of the CNODE contraction. In all other cases the CNODE is simply returned as-is. A CONTRACTION represents the first of the two-step *ARC RETRACTION PROTOCOL* that effects the reclaimation of allocated storage no longer used and the optimization of of lookup efficiency by compacting CTRIE depth and thereby the number of levels which must be traversed.

For further information, refer to the function `CNODE-COMPRESSED` which implements the second stage of this protocol, completing the process.

[function] `CNODE-COMPRESSED (CNODE LEVEL)`

The *COMPRESSION* of a CNODE is the second step of the *ARC RETRACTION PROTOCOL* completing a retraction that has been initiated by `CNODE-CONTRACTED`. The CNODE compression is computed by generating a replacement cnode structure that is similar to CNODE, but with any entombed inode arcs created during contraction simply replaced by the SNODE that had been entombed. This is called the *RESURRECTION* of that SNODE. After all entombed inode arcs of a cnode have been collapsed into simple SNODE leaves, if the resulting CNODE has been compressed so far as to contain only a single SNODE leaf, it is subjected to another CONTRACTION before it is returned as the result of the compression. Otherwise it is simply returned and represents a complete iteration of the *ARC RETRACTION PROTOCOL*

[function] `CLEAN (INODE LEVEL)`

CLEAN is the basic entry-point into the arc retraction protocol. Given an arbitrary, non-root inode referencing a CNODE that can be compressed, update that inode to reference the result of that compression. Otherwise INODE will remain unaffected.

[function] `CLEAN-PARENT (PARENT-INODE TARGET-INODE KEY LEVEL)`

During a `CTRIE-DROP` (`%remove`) operation, if the result of a KEY/VALUE removal is an arc consisting of an `ENTOMBED` inode (one referencing a TNODE), then, if that arc remains accessible from the parent of a CNODE containing it, generate the compression of that CNODE and update its parent INODE with the result.

[structure] `RDCSS-DESCRIPTOR ()`

An RDCSS-DESCRIPTOR object represents a 'plan' for a proposed RDCSS (restricted double compare single swap) operation. The use of this descriptor object provides the means to effect an atomic RDCSS in software, requiring only hardware support for single-word CAS, which is preferable because it is commonly available on curent consumer hardware.

- `OV` designates a slot containing the OLD (current) root inode. If the swap is unsuccessful, the resulting ctrie will revert to this structure as the root inode.
- `OVMAIN` designates a slot containing the CNODE that is referenced by the OLD (current) root inode.
- `NV` designates a slot containing a fully assembled replacement root inode referencing a valid CNODE. This pair will become the root inode and level 0 MAIN-NODE of the ctrie if the swap is successful.
- `COMMITTED` designates a flag which, when not NIL, indicates that the RDCSS plan defined by this descriptor has completed successfully

[function] `RDCSS-DESCRIPTOR-P (OBJECT)`

Returns T if the specified object is of type rdcss-descriptor.

[function] `RDCSS-DESCRIPTOR-OV (RDCSS-DESCRIPTOR)`

Returns the ov of the specified rdcss-descriptor

[function] `RDCSS-DESCRIPTOR-OVMAIN (RDCSS-DESCRIPTOR)`

Returns the ovmain of the specified rdcss-descriptor

[function] `RDCSS-DESCRIPTOR-NV (RDCSS-DESCRIPTOR)`

Returns the nv of the specified rdcss-descriptor

[function] `RDCSS-DESCRIPTOR-COMMITTED (RDCSS-DESCRIPTOR)`

Returns and (with setf) changes the committed of the specified rdcss-descriptor

[function] `ROOT-NODE-ACCESS (CTRIE &OPTIONAL ABORT)`

ROOT-NODE-ACCESS extends `FIND-CTRIE-ROOT`, implementing the *RDCSS ROOT NODE PROTOCOL* for access to root inode of CTRIE. In particular, it ensures that if, instead of an inode, the root of CTRIE contains an RDCSS descriptor of a proposed root-node update, that it will immediately invoke `ROOT-NODE-COMMIT` to act on that descriptor and return an INODE struct that is the result of the completed commit process. `ROOT-NODE-ACCESS` only provides access to the root inode *STRUCTURE* and in particular it does not provide safe access to the *CONTENT* of that inode. In order to access those contents, the root inode returned by `ROOT-NODE-ACCESS` must be further processed by `INODE-READ` in order to still properly comply with the underlying GCAS protocol implementation requirements common to all inodes

[function] `ROOT-NODE-REPLACE (CTRIE OV OVMAIN NV)`

ROOT-NODE-REPLACE implements the *RDCSS ROOT NODE PROTOCOL* for replacement of the ROOT INODE of a CTRIE structure with another one that contains new or alternative values, achieving the end-result effectively the same as if by mutation. The replacement of the root inode is accomplished in two, basic conceptual stages. First, an `RDCSS-DESCRIPTOR` object which specifies in full the current state and all desired changes in the proposed resulting state. Thus, whether any individual replacement attempt succeeds or fails, either result is guarenteed to represent a valid state. An attempt is then made to atomically swap this RDCSS-DESCRIPTOR with the current CTRIE root inode. Note that, although it contains all of the information representing two, distinct, root inode states, the RDCSS-DESCRIPTOR is not, itself, a valid root inode. That is the reason why all access to the root inode must be accomplished using this specialized *RDCSS ROOT NODE PROTOCOL* rather than just the *GCAS INODE PROTOCOL* alone, as is done with all other non-root inodes. Once an atomic compare-and-swap of an RDCSS-DESCRIPTOR object with the root inode completes successfully, `ROOT-NODE-COMMIT` is invoked which will attempt to complete the second step of this protocol. The result of that commit will be one or the other of the two valid states defined in the RDCSS-DESCRIPTOR object. If another thread concurrently attempts access to a root node holding an RDCSS-DESCRIPTOR object rather than an INODE, it will invoke `ROOT-NODE-COMMIT` itself, possibly prempting our own attempt, but guaranteeing nonblocking access to a valid root node by any concurrent thread

[function] `ROOT-NODE-COMMIT (CTRIE &OPTIONAL ABORT)`

rdcss api to complete a root-node transaction *[function]* `CTRIE-SNAPSHOT (CTRIE &KEY READ-ONLY)`

[function] `CTRIE-CLEAR (CTRIE)`

[function] `CTRIE-PUT (CTRIE KEY VALUE)`

Insert a new entry into CTRIE mapping KEY to VALUE. If an entry with key equal to KEY aleady exists in CTRIE, according to the equality predicate defined by `CTRIE-TEST` then the priorbmapping will be replaced by VALUE. Returns `VALUE` representing the mapping in the resulting CTRIE

[function] `%INSERT (INODE KEY VALUE LEVEL PARENT STARTGEN)`

- The detailed specifics required to perform an insertion into a CTRIE map are defined by and contained within the `%INSERT` function, which is not part of the USER API and should never be invoked directly. The procedures required for interaction with `%INSERT` are managed entirely by the upper layer entry-points and should be entirely invisible to the developer using CL-CTRIE. The alogorithm is intricate and requires quite some effort to figure out based on the papers and documentation. For that reason this attempt is made to properly document the process -- not to encourage anyone to fiddle with it...

1. A given call to %insert carries with it no guarantee that it will actually succeed. Further, there is no guarantee it will do anything at all -- including ever returning to the caller having invoked it. This is because there are a number of circumstances that may possibly interfere with insertion in a lock-free concurrent data-structure and in order to minimize the cost incurred by aborted attempts and restarts, there is no effort wasted on careful condition handling or recovery. Instead, as soon as it is recognised that a particular attempt will not succeed, Control is thrown via non-local exit unceremoneously the entire call stack and restarting each time from the very beginning. There are quite a few places within the process where you will find these nonlocal exits. Originally I had in mind to incorporate some additional insrumentation to track and gather statistics the log the specifics of all this, and there are a number of extension points provided. For more details refer to the `MULTI-CATCH` and `CATCH-CASE` control structure documentation.

2. %insert ALWAYS begins at an INODE. This is not surprising, of course, since the root of the tree (where all inserts begin) is an INODE, and because INODES are the only nodes that provide mutability. When we which to effect change to the CTRIE regardless of where or what type, It may only be accomplished by mutating the parent INODE immediately above it to reference content that must be freshly regenerated each time changes are required. Once it is established that the %insert always begins at an inode, we can reason further about the sequence of events that follow by considering some of the node oriented invariants for CTRIEs specified variously in the academic literature. First, it has been clearly defined that an INODE structure may only reference three kinds of structure that we collectively refer to as MAIN-NODES. These are the three potential cases which we will consider next.

3. Consider first the TNODE. If indeed we follow an INODE and discover it directly leads us to a TNODE, or 'Tomb Node'. This tells us, first, that we have arrived at a dead-end, second that we must assist with the 'compression' of this arc by invoking the `CLEAN` operation on the tombed INODE's parent. Finally, there is nothing further we can do so we THROW to :RESTART.

4. If traverse the INODE and arrive at an LNODE, we are also at the end of the ARC, but if it is due to hash collision then the algorithm then indeed it may be correct. In this case we attempt to 'insert' ourself in te LNODE chain and then invoke INODE mutate to atomic commit and then THROW to :RESTART

5. As a simple instance of the general case, we may arrive at a CNODE with vacant arc that represents the index specified by the bits of our KEY's hash code that are active for this level within the ctrie. When this is the case, we construct a replacement CNODE augmented with our key/value pair as an SNODE LEAF at the physical position within the CNODES storage vector appropriately translated from the logical arc index as described by the documentation of the functions `FLAG-ARC-POSITION` and `FLAG-VECTOR` If we successfully mutate the parent inode by completing an atomic replacement of the old cnode with the one we constructed, then our insertion has succeeded and we return the range value now successfully mapped by KEY in order to indicate our success. Otherwise we THROW to :RESTART.

6. If we find the logical index of our 'arc' in this CNODE is not empty and available as we did above, there are exactly two other possibly find there; we know this because it is required to be a BRANCH-NODE -- either an snode leaf storage or an inode referencing a MAIN-NODE that represents the next layer of the CTRIE. We describe various posible cases and the define a procedure specified for each below.

7. If we find that the node PRESENT at this index is an INODE, then this is the simplest of the possible cases. Conceptually, what we intend to do is continue to follow our arc, descending to the next level of the CTRIE structure that is referenced by that inode. In practice, however, we are required to consider the possibility that the generational descriptor object the inode contains may not be consistent with STARTGEN, which is the one current in the root INODE of this CTRIE. This may be the case, for example, as the result of some past cloning/snapshot operation if we are the first since then to traverse this inode. (Remember that the refresh of generational descriptor occurs lazily on an as-needed basis in order to avoid overhead incurred by eager traversals which often turn out to have been unnecessary). In consideration of this we proceeed as follows: - If the inode generational descriptor is consistent with STARTGEN, we simply continue along our arc by recursively invoking `%insert` on that INODE. If that function call returns successfully with a VALUE, then the insertion was successful, and we also then return, passing along that value. Thus the result is communicated back through the caller chain, eventually arriving back as the result of the original CTRIE-PUT entry-point. - Otherwise if the generational descriptor is not consistent with STARTGEN we attempt an atomic `INODE-MUTATE` on the PARENT INODE of this CNODE that effects the replacement of that inode within it by one FRESHLY CREATED by `REFRESH` and ensured to be consistent with STARTGEN. If this succeeds, we invoke %insert recusively and proceed in the same manner, since, effectively, we are now in a state equivalent to the one described above. - If the INODE-MUTATE of the prior step did NOT succeed, then we are out of options and throw to :RESTART the insertion process from the beginning all over again.

8. If we find that the node PRESENT at this index is an SNODE, then our situation becomes a little bit more complex and there are a few more contingencies we must be prepared to address.

9. Once again, looking at the simplest first, when an insert operation encounters a leaf-node somewhere along the descent of it's 'own' arc, one potential case is that it found the node it was looking for -- one that contains a key that satisfies the test predicate defined for the dy-

namic extent of the current operation, `CTEQUAL,` when compared to the `KEY` currently being `%INSERTED.` If the equality test is satisfied then the VALUE that node maps should be updated with the one of the present insertion. The steps to effect the update are very similar to those of step 5, however We construct a replacement CNODE augmented with our key/value pair as a replacement SNODE in the SAME physical position as the one we have found -- refer to the documentation for the function `CNODE-UPDATED` for additional specifics on the internal details that describe this operation. If we successfully mutate the parent inode by completing an atomic replacement of the old cnode with the one we constructed, then our update has succeeded and we return the range value now successfully mapped by KEY in order to indicate our success. Otherwise we THROW to :RESTART.

10. In some circumstances, we encounter a node on our arc whose hash code bits have matched that of the current key of this insertion for all of the lower order bits that have been consumed so far, up to the current depth, but that (as opposed to step 9) does not satisfy `CTEQUAL.` and so is NOT a candidiate for update replacement. Except in very vare circumstances, there will be some depth at which the active bits of its hash code will indeed be distinct from our own, and at that point a CNODE can be constructed that will proprerly contain both it and an snode mapping the key/value of the current insertion. This means we must ENTEND the ctrie as many layers as needed to get to that depth, inserting CNODES and INODES at each step along the way. Now, we will first describe the 'edge' case where we have encounted the 'rare circumstance.' If we perform this process and arrive at a depth where all 32 hash code bits have been consumed and, indeed, these two unequal keys are the result of a 'hash code collision' In order that we preserve correct operation, we respond in this case by chaining these key/value SNODES into a linked list of LNODES. Therefore, they can share the same arc index and when we encounter such a thing during future traversals, we can accomodate the collision using simple linear search and a few basic LNODE utility functions such as `LNODE-INSERTED` `LNODE-REMOVED` `LNODE-SEARCH` `LNODE-LENGTH` and the list constructor `ENLIST.` Once we have `ENLIST`ed the colliding SNODES, we create a new INODE pointing to that list, and then attempt atomic replacement of the CNODE above with one we extend to contain that INODE. If we do successfully mutate the prior CNODES parent INODE resulting in its replacement with the CNODE we constructed, then our insert has succeeded and we return the range value now successfully mapped by KEY in order to indicate our success. Otherwise we THROW to :RESTART.

11. Finally, let us return to those intermediate steps, mentioned above, to specify the means by which we perform the level-by-level extension of a given arc to accommodate both the above case of hash collision as well as the more common one when the reason for extension is simply to accomodate normal growth capacity and allocation. In both cases, though, the extensions are performed for the same initiating cause -- to accomodate the collision of leaf node keys resident at lower levels of the structure. Depending on the similarity of two colliding hash keys, the extension process may not be resolved with a single iteration. In the case of full collisiion, described above, the extension process will recur, up to a maximum depth of `(32/W)` levels, at which point an L-NODE chain will be created. At each iteration, a new INODE is created, pointing to a new CNODE containing one of our conflictung pairs. Then, `%INSERT` is attempted on that INODE and this process recurs. Once this cycle of insert/extend completes, each INODE/CNODE pair is returned to the parent -- the entire newly created structure eventually returning to the point of original conflicts whre the extension cycle began. If we successfully mutate the parent inode by completing an atomic replacement of the

old cnode with the one that begins this newly built structue, then our update has succeeded and we return the range value now successfully mapped by KEY in order to indicate our success. Otherwise we THROW to :RESTART

[function] `CTRIE-GET (CTRIE KEY)`

[function] `%LOOKUP (INODE KEY LEVEL PARENT STARTGEN)`

The general concept of the procedure for finding a given key within a simplified SEQUENTIAL model of a CTRIE can be summarized as follows: If the internal node is at level `L` then the W bits of the hashcode starting from position `W * L` are used as a logical index into the vector of `2^W` arcs that can possibly be represented within that node (see `FLAG` and `FLAG-VECTOR`). This logical index is then transformed into a physical index that denotes a specific position locating this arc relative to all other arcs currently present in the node (see `FLAG-ARC-POSITION`. In this way, storage within the node need not be allocated for representation of empty arc positions. At all times the invariant is maintained that the number of arcs allocated within a given CNODE is equal to the Hamming-Weight of its BITMAP -- i.e., the number of nonzero bits present (see `CL:LOGCOUNT`). The arc at this calculated relative position is then followed, and the process repeated until arrival at a leaf-node or empty arc position. Locating a given key becomes substantially more complicated in the actual lock-free concurrent ctrie algorithm

[function] `CTRIE-DROP (CTRIE KEY)`

Remove KEY and it's value from the CTRIE.

[function] `%REMOVE (INODE KEY LEVEL PARENT STARTGEN)`

[function] `CTRIE-MAP (CTRIE FN &KEY ATOMIC &AUX ACCUM)`

[macro] `CTRIE-DO ((KEY VALUE CTRIE &KEY ATOMIC) &BODY BODY)`

Iterate over (key . value) in ctrie in the manner of dolist. ;;; EXAMPLE: (ctrie-do (k v ctrie) ;;; (format t "~&~8S => ~10S~%" k v))

[function] `CTRIE-MAP-KEYS (CTRIE FN &KEY ATOMIC)`

[function] `CTRIE-MAP-VALUES (CTRIE FN &KEY ATOMIC)`

[function] `CTRIE-KEYS (CTRIE &KEY ATOMIC)`

[function] `CTRIE-VALUES (CTRIE &KEY ATOMIC)`

[function] `CTRIE-SIZE (CTRIE &AUX (ACCUM 0))`

[function] `CTRIE-EMPTY-P (CTRIE)`

[function] `CTRIE-TO-ALIST (CTRIE &KEY ATOMIC)`

[function] `CTRIE-TO-HASHTABLE (CTRIE &KEY ATOMIC)`

[function] `CTRIE-PPRINT (CTRIE &OPTIONAL (STREAM T))`

[function] `CTRIE-FROM-ALIST (ALIST &KEY CTRIE)`

[function] `CTRIE-FROM-HASHTABLE (HASHTABLE &KEY CTRIE)`

create a new ctrie containing the same (k . v) pairs and equivalent test function as HASHTABLE

[macro] `DEFINE-CTRIE (NAME CTRIE &REST ARGS &KEY (OBJECT T) SPEC)`

Define a 'functional' **CTRIE-LAMBDA** that combines all the the capabilities of the raw data structure with behavior and semantics one would expect of any other ordinary common-lisp function. The resulting symbol defined as 'name will be bound in three distinct namespaces: the `SYMBOL-VALUE` will be bound to the LAMBDA CLOSURE object, `SYMBOL-FUNCTION` (fdefinition) will be FBOUND to the compiled function, and the corresponding '(SETF NAME) form will be SETF-BOUND. the syntax for invoking NAME is as in a LISP1; i.e., no 'funcall' is required (but still works if you prefer). Calling `(NAME key)` returns the value mapped to key, or `NIL` just as if by `(CTRIE-GET ctrie-name key)`. Analogously when used as a setf-able place such as by `(setf (NAME key) value)` it has the equivalent behavior to the operation `(CTRIE-PUT ctrie-name key value)`. Use of this type of binding technique has some really convenient effects that I've quickly started to become quite fond of. One such idiom, for example, `(mapcar MY-CTRIE '(key1 key2 key3 key4 ...))` returns a list containing all the mapped values corresponding to the respective keys. One additional feature that I've found extremely useful is included *under the hood:* Invoking MY-CTRIE on an object of type FUNCTION will not search the ctrie for an entry having that function ast its key, but will instead APPLY that function to the actual CTRIE structure wrapped within the closure. Thus, `(MY-CTRIE #'identity)` will return the underlying ctrie as just an ordinary instance of a CTRIE STRUCTURE.
There are many other functions this is handy with, like `(MY-CTRIE #'ctrie-size)` `(MY-CTRIE #'ctrie-to-hashtable)` etc. Some additional examples are provided below.

```
  ;;;  (define-ctrie my-ctrie)
  ;;;    =>  MY-CTRIE
  ;;;
  ;;;  (describe 'my-ctrie)
  ;;;
  ;;;     CL-CTRIE::MY-CTRIE
  ;;;       [symbol]
  ;;;
  ;;;     MY-CTRIE names a special variable:
  ;;;       Value: #<CLOSURE (LAMBDA # :IN MAKE-CTRIE-LAMBDA)
{100F73261B}>
  ;;;
  ;;;     MY-CTRIE names a compiled function:
  ;;;       Lambda-list: (&REST ARGS1)
  ;;;       Derived type: FUNCTION
  ;;;
  ;;;     (SETF MY-CTRIE) names a compiled function:
  ;;;       Lambda-list: (VALUE KEY)
  ;;;       Derived type: (FUNCTION (T T) *)
  ;;;
```

```
;;;
;;;   (my-ctrie :HONG-KONG :FOOY)
;;;     =>  :FOOY
;;;
;;;   (my-ctrie :HONG-KONG)
;;;     =>  :FOOY ; T
;;;
;;;   (map 'list #'eval (mapcar #`(my-ctrie ,a1 ,a1) (iota 12)))
;;;     =>  (0 1 2 3 4 5 6 7 8 9 10 11)
;;;
;;;   (mapcar my-ctrie (iota 12))
;;;     =>  (0 1 2 3 4 5 6 7 8 9 10 11)
```

[generic-function] `CTRIE-LAMBDA-DISPATCH (CTRIE-LAMBDA-OBJECT)`

Returns and (with setf) changes the dispatch of the specified ctrie-lambda-object

[function] `CTRIE-LAMBDA-SPAWN (SELF &KEY READ-ONLY)`

Causes the atomic clone of enclosed ctrie structure and builds a new lexical closure to operate on it. Does not bother to reproduce fancy (expensive) object, class, bindings, but provides almost identical functionality. May be used to more efficintly distribute workload in parallel

[macro] `CTRIE-LAMBDA (&ONCE CTRIE &REST REST)`

Pandoric Object and Inter-Lexical Communication Protocol this macro builds the most unencumbered and widely applicable 'purist edition' Of our PLAMBDA based form. Even as such, a lot of care has been given to many subtle ways it has been refined to offer the most convenient and natural tool possible.

```
     ;;; (plambda (#<CLOSURE (LAMBDA (&REST ARGS)) {100929EB1B}> )
     ;;;
     ;;; DISPATCHING to FUNCTIONAL MAPPING:
     ;;;   (IF (REST ARGS)
     ;;;          (APPLY ARG (REST ARGS))
     ;;;          (FUNCALL ARG #'IDENTITY)) =>
     ;;; ------------------------------------------------------------
     ;;; INITIALIZING PLAMBDA
     ;;; ------------------------------------------------------------
     ;;;   IT => #S(CTRIE
     ;;;                :READONLY-P NIL
     ;;;                :TEST EQUAL
     ;;;                :HASH SXHASH
     ;;;                :STAMP #<CLOSURE (LAMBDA # :IN CONSTANTLY)
{10092B516B}>
     ;;;                :ROOT #S(INODE
     ;;;                         :GEN #:|ctrie2196|
     ;;;                         :REF #S(REF
     ;;;                                 :STAMP @2012-08-
19T13:34:58.314457-04:00
```

```
    ;;;                                 :VALUE #S(CNODE :BITMAP 0 :ARCS #
())
    ;;;                                 :PREV NIL)))
    ;;;   PLIST => (:CONTAINER #<CLOSURE (LAMBDA #) {100929EACB}>
    ;;;             :TIMESTAMP @2012-08-19T13:34:58.314464-04:00)
    ;;;   STACK => (#<CLOSURE (LAMBDA #) {100929EACB}>)
    ;;;
    ;;; ------------------------------------------------------------
    ;;;  #<CLOSURE (LAMBDA (&REST #:ARGS55)) {100929EACB}>
    ;;;```
```

[generic-function] `CTRIE-LAMBDA-DISPATCH (CTRIE-LAMBDA-OBJECT)`

Returns and (with setf) changes the dispatch of the specified ctrie-lambda-object

[special-variable] `+SIMPLE-DISPATCH+ ((DLAMBDA (:FROM (ARG) ARG) (:TO (ARG) ARG) (:DOMAIN (ARG) ARG) (:RANGE (ARG) ARG)))`

[macro] `CTRIE-ERROR (CONDITION &REST ARGS)`

Signal a CTRIE related condition.

[condition] `CTRIE-ERROR (ERROR)`

Abstract superclass of CTRIE related conditions.

[condition] `CTRIE-STRUCTURAL-ERROR (CTRIE-ERROR)`

Condition designating that the CTRIE data structure has been determined to be invalid.

[condition] `CTRIE-OPERATIONAL-ERROR (CTRIE-ERROR)`

Condition for when an operational failure or inconsistency has occurred.

[function] `CTRIE-MODIFICATION-FAILED (REASON &KEY OP PLACE)`

Signal a modification failure with the appropriate attendant metadata.

[condition] `CTRIE-MODIFICATION-FAILED (CTRIE-OPERATIONAL-ERROR)`

This condition indicates an unhandled failure of an attempt to perform stateful modification to CTRIE. The most common case in which this might occur is when such an attempt is mode on a CTRIE designated as READONLY-P. In any case, this condition represents an exception from which processing cannot continue and requires interactive user intervention in order to recover.

[condition] `CTRIE-OPERATION-RETRIES-EXCEEDED (CTRIE-OPERATIONAL-ERROR)`

Condition indicating an operation has failed the maximum number of times specified by the special-variable *retries*

[condition] `CTRIE-NOT-IMPLEMENTED (CTRIE-ERROR)`

Condition designating functionality for which the implementation has not been written, but has not been deliberately excluded.

[condition] `CTRIE-NOT-SUPPORTED (CTRIE-ERROR)`

Condition designating functionality that is deliberately not supported.

[condition] `CTRIE-INVALID-DYNAMIC-CONTEXT (CTRIE-OPERATIONAL-ERROR)`

Condition indicating an operation was attempted outside the dynamic extent of a valid enclosing WITH-CTRIE form

[condition] `CTRIE-GENERATIONAL-MISMATCH (CTRIE-STRUCTURAL-ERROR)`

Condition indicating an operation encountered an outdated or inconsistent node during its attempted traversal

[function] `README (&OPTIONAL (STREAM *STANDARD-OUTPUT*))`

Update documentation sections of the README file. When an output stream is specified, the results are also echoed to that stream. To inhibit output, invoke as `(readme (make-broadcast-stream))` or use `README-QUIETLY`

[function] `README-QUIETLY ()`

Update documentation sections of the README file, inhibiting any other printed output.

[function] `APIDOC (&OPTIONAL (SCOPE :EXTERNAL))`

Collect a list of strings representing the documentation for CL-CTRIE rendered in a compact format suitable for inclusion in a lightweight text-markup format document. If SCOPE is specified it must be either :EXTERNAL. corresponding to those symbols exported as the public API, or :HOME, which designates all symbols defined locally in package.

[function] `PRINC-APIDOC (&OPTIONAL (SCOPE :EXTERNAL))`

Print to `*STANDARD-OUTPUT*` the documentation for CL-CTRIE rendered in a compact format. This is intended primarily as a convenience to the interactive user seeking quick reference at the REPL. If SCOPE is specified it must be either :EXTERNAL. corresponding to those symbols exported as the public API, or :HOME, which designates all symbols defined locally in package.

[function] `COLLECT-DOCS (&OPTIONAL (SCOPE :EXTERNAL) (SORT #'STRING<))`

Regenerate on-disk html documentation and collect the cached in-memory descriptors for further processing. If SCOPE is specified it must be either :EXTERNAL. corresponding to those symbols exported as the public API, or :HOME, which designates all symbols defined locally in package. Output order may be customized by an optionally specified SORT function.

[macro] `DEFINE-DIAGRAM (TYPE (&OPTIONAL CONTEXT) &BODY BODY)`

Define a diagrammatic representation of TYPE, optionally specialized for a specific CONTEXT. See {defgeneric cl-ctrie::make-diagram}.