nim-lang.org

Nim Destructors and Move Semantics

19–24 minutes

Source Edit

Authors:	Andreas Rumpf
Version:	2.2.4

About this document

This document describes the ARC/ORC Nim runtime which does not use classical GC algorithms anymore but is based on destructors and move semantics. The advantages are that Nim programs become oblivious to the involved heap sizes and programs are easier to write to make effective use of multi-core machines. As a nice bonus, files and sockets and the like can be written not to require manual `close` calls anymore.

This document aims to be a precise specification about how move semantics and destructors work in Nim.

Motivating example

With the language mechanisms described here, a custom seq could be written as:

```
type
  myseq*[T] = object
```

```
  len, cap: int
  data: ptr UncheckedArray[T]

proc `=destroy`*[T](x: myseq[T]) =
 if x.data != nil:
  for i in 0..<x.len: `=destroy`(x.data[i])
  dealloc(x.data)

proc `=wasMoved`*[T](x: var myseq[T]) =
 x.data = nil

proc `=trace`[T](x: var myseq[T]; env: pointer) =


 if x.data != nil:
  for i in 0..<x.len: `=trace`(x.data[i], env)

proc `=copy`*[T](a: var myseq[T]; b: myseq[T]) =

 if a.data == b.data: return
 `=destroy`(a)
 `=wasMoved`(a)
 a.len = b.len
 a.cap = b.cap
 if b.data != nil:
  a.data = cast[typeof(a.data)](alloc(a.cap * sizeof(T)))
  for i in 0..<a.len:
   a.data[i] = b.data[i]

proc `=dup`*[T](a: myseq[T]): myseq[T] {.nodestroy.} =
```

```
  result = myseq[T](len: a.len, cap: a.cap, data: nil)
  if a.data != nil:
    result.data = cast[typeof(result.data)](alloc(result.cap * sizeof(T)))
    for i in 0..<result.len:
      result.data[i] = `=dup`(a.data[i])

proc `=sink`*[T](a: var myseq[T]; b: myseq[T]) =


  `=destroy`(a)
  a.len = b.len
  a.cap = b.cap
  a.data = b.data

proc add*[T](x: var myseq[T]; y: sink T) =
  if x.len >= x.cap:
    x.cap = max(x.len + 1, x.cap * 2)
    x.data = cast[typeof(x.data)](realloc(x.data, x.cap * sizeof(T)))
  x.data[x.len] = y
  inc x.len

proc `[]`*[T](x: myseq[T]; i: Natural): lent T =
  assert i < x.len
  x.data[i]

proc `[]=`*[T](x: var myseq[T]; i: Natural; y: sink T) =
  assert i < x.len
  x.data[i] = y
```

```
proc createSeq*[T](elems: varargs[T]): myseq[T] =
 result = myseq[T](
  len: elems.len,
  cap: elems.len,
  data: cast[typeof(result.data)](alloc(result.cap * sizeof(T))))
 for i in 0..<result.len: result.data[i] = elems[i]

proc len*[T](x: myseq[T]): int {.inline.} = x.len
```

Lifetime-tracking hooks

The memory management for Nim's standard `string` and `seq` types as well as other standard collections is performed via so-called "Lifetime-tracking hooks", which are particular type bound operators.

There are 6 different hooks for each (generic or concrete) object type `T` (`T` can also be a `distinct` type) that are called implicitly by the compiler.

(Note: The word "hook" here does not imply any kind of dynamic binding or runtime indirections, the implicit calls are statically bound and potentially inlined.)

`=destroy` hook

A `=destroy` hook frees the object's associated memory and releases other associated resources. Variables are destroyed via this hook when they go out of scope or when the routine they were declared in is about to return.

A `=destroy` hook is allowed to have a parameter of a `var T` or `T` type. The prototype of this hook for a type `T` needs to be:

proc `=destroy`(x: T)

The general pattern in `=destroy` looks like:

```
proc `=destroy`(x: T) =

 if x.field != nil:
  freeResource(x.field)
```

A `=destroy` is implicitly annotated with `.raises: []`; a destructor should not raise exceptions. For backwards compatibility the compiler produces a warning for a `=destroy` that does raise.

A `=destroy` can explicitly list the exceptions it can raise, if any, but this of little utility as a raising destructor is implementation defined behavior. Later versions of the language specification might cover this case precisely.

`=wasMoved` hook

A `=wasMoved` hook sets the object to a state that signifies to the destructor there is nothing to destroy.

The prototype of this hook for a type `T` needs to be:

proc `=wasMoved`(x: var T)

Usually some pointer field inside the object is set to `nil`:

```
proc `=wasMoved`(x: var T) =
 x.field = nil
```

`=sink` hook

A `=sink` hook moves an object around, the resources are stolen from the source and passed to the destination. It is ensured that the source's destructor does not free the resources afterward by setting the object to its

default value (the value the object's state started in). Setting an object `x` back to its default value is written as `wasMoved(x)`. When not provided the compiler is using a combination of `=destroy` and `copyMem` instead. This is efficient hence users rarely need to implement their own `=sink` operator, it is enough to provide `=destroy` and `=copy`, the compiler will take care of the rest.

The prototype of this hook for a type `T` needs to be:

```
proc `=sink`(dest: var T; source: T)
```

The general pattern in `=sink` looks like:

```
proc `=sink`(dest: var T; source: T) =
  `=destroy`(dest)
  wasMoved(dest)
  dest.field = source.field
```

Note: `=sink` does not need to check for self-assignments. How self-assignments are handled is explained later in this document.

`=copy` hook

The ordinary assignment in Nim conceptually copies the values. The `=copy` hook is called for assignments that couldn't be transformed into `=sink` operations.

The prototype of this hook for a type `T` needs to be:

```
proc `=copy`(dest: var T; source: T)
```

The general pattern in `=copy` looks like:

```
proc `=copy`(dest: var T; source: T) =

  if dest.field != source.field:
```

```
`=destroy`(dest)
wasMoved(dest)
dest.field = duplicateResource(source.field)
```

The `=copy` proc can be marked with the `{.error.}` pragma. Then any assignment that otherwise would lead to a copy is prevented at compile-time. This looks like:

```
proc `=copy`(dest: var T; source: T) {.error.}
```

but a custom error message (e.g., `{.error: "custom error".}`) will not be emitted by the compiler. Notice that there is no `=` before the `{.error.}` pragma.

`=trace` hook

A custom **container** type can support Nim's cycle collector `--mm:orc` via the `=trace` hook. If the container does not implement `=trace`, cyclic data structures which are constructed with the help of the container might leak memory or resources, but memory safety is not compromised.

The prototype of this hook for a type `T` needs to be:

```
proc `=trace`(dest: var T; env: pointer)
```

`env` is used by ORC to keep track of its internal state, it should be passed around to calls of the built-in `=trace` operation.

Usually there will only be a need for a custom `=trace` when a custom `=destroy` that deallocates manually allocated resources is also used, and then only when there is a chance of cyclic references from items within the manually allocated resources when it is desired that `--mm:orc` is able to break and collect these cyclic referenced resources. Currently however, there is a mutual use problem in that whichever of `=destroy`/`=trace` is

used first will automatically create a version of the other which will then conflict with the creation of the second of the pair. The workaround for this problem is to forward declare the second of the "hooks" to prevent the automatic creation.

The general pattern in using `=destroy` with `=trace` looks like:

```
type
 Test[T] = object
  size: Natural
  arr: ptr UncheckedArray[T]

proc makeTest[T](size: Natural): Test[T] =
 Test[T](size: size, arr: cast[ptr UncheckedArray[T]](alloc0(sizeof(T) *
size)))


proc `=destroy`[T](dest: Test[T]) =
 if dest.arr != nil:
  for i in 0 ..< dest.size: dest.arr[i].`=destroy`
  dealloc dest.arr

proc `=trace`[T](dest: var Test[T]; env: pointer) =
 if dest.arr != nil:

  for i in 0 ..< dest.size: `=trace`(dest.arr[i], env)
```

Note: The `=trace` hooks (which are only used by `--mm:orc`) are currently more experimental and less refined than the other hooks.

`=dup` hook

A `=dup` hook duplicates an object. `=dup(x)` can be regarded as an optimization replacing a `wasMoved(dest); =copy(dest, x)` operation.

The prototype of this hook for a type `T` needs to be:

```
proc `=dup`(x: T): T
```

The general pattern in implementing `=dup` looks like:

```
type
 Ref[T] = object
  data: ptr T
  rc: ptr int

proc `=dup`[T](x: Ref[T]): Ref[T] =
 result = x
 if x.rc != nil:
  inc x.rc[]
```

Move semantics

A "move" can be regarded as an optimized copy operation. If the source of the copy operation is not used afterward, the copy can be replaced by a move. This document uses the notation `lastReadOf(x)` to describe that `x` is not used afterward. This property is computed by a static control flow analysis but can also be enforced by using `system.move` explicitly.

One can query if the analysis is able to perform a move with `system.ensureMove.move` enforces a move operation and calls `=wasMoved` whereas `ensureMove` is an annotation that implies no runtime operation. An `ensureMove` annotation leads to a static error if the compiler cannot prove that a move would be safe.

For example:

```
proc main(normalParam: string; sinkParam: sink string) =
 var x = "abc"

 let valid = ensureMove x

 let invalid = ensureMove normalParam

 let alsoValid = ensureMove sinkParam
```

Swap

The need to check for self-assignments and also the need to destroy previous objects inside `=copy` and `=sink` is a strong indicator to treat `system.swap` as a builtin primitive of its own that simply swaps every field in the involved objects via `copyMem` or a comparable mechanism. In other words, `swap(a, b)` is **not** implemented as `let tmp = move(b); b = move(a); a = move(tmp).`

This has further consequences:

- Objects that contain pointers that point to the same object are not supported by Nim's model. Otherwise swapped objects would end up in an inconsistent state.
- Seqs can use `realloc` in the implementation.

Sink parameters

To move a variable into a collection usually `sink` parameters are involved. A location that is passed to a `sink` parameter should not be used afterward. This is ensured by a static analysis over a control flow graph. If it

cannot be proven to be the last usage of the location, a copy is done instead and this copy is then passed to the sink parameter.

A sink parameter *may* be consumed once in the proc's body but doesn't have to be consumed at all. The reason for this is that signatures like `proc put(t: var Table; k: sink Key, v: sink Value)` should be possible without any further overloads and `put` might not take ownership of `k` if `k` already exists in the table. Sink parameters enable an affine type system, not a linear type system.

The employed static analysis is limited and only concerned with local variables; however, object and tuple fields are treated as separate entities:

```
proc consume(x: sink Obj) = discard "no implementation"

proc main =
  let tup = (Obj(), Obj())
  consume tup[0]

  echo tup[1]
```

Sometimes it is required to explicitly `move` a value into its final position:

```
proc main =
  var dest, src: array[10, string]

  for i in 0..high(dest): dest[i] = move(src[i])
```

An implementation is allowed, but not required to implement even more move optimizations (and the current implementation does not).

Sink parameter inference

The current implementation can do a limited form of sink parameter

inference. But it has to be enabled via `--sinkInference:on`, either on the command line or via a `push` pragma.

To enable it for a section of code, one can use `{.push sinkInference: on.}` ... `{.pop.}`.

The .nosinks pragma can be used to disable this inference for a single routine:

```
proc addX(x: T; child: T) {.nosinks.} =
 x.s.add child
```

The details of the inference algorithm are currently undocumented.

Rewrite rules

Note: There are two different allowed implementation strategies:

1. The produced `finally` section can be a single section that is wrapped around the complete routine body.
2. The produced `finally` section is wrapped around the enclosing scope.

The current implementation follows strategy (2). This means that resources are destroyed at the scope exit.

```
var x: T; stmts
--------------          (destroy-var)
var x: T; try stmts
finally: `=destroy`(x)



g(f(...))
----------------------  (nested-function-call)
g(let tmp;
```

```
bitwiseCopy tmp, f(...);
tmp)
finally: `=destroy`(tmp)
```

```
x = f(...)
-----------------------   (function-sink)
`=sink`(x, f(...))
```

```
x = lastReadOf z
------------------        (move-optimization)
`=sink`(x, z)
`=wasMoved`(z)
```

```
v = v
------------------  (self-assignment-removal)
discard "nop"
```

```
x = y
------------------        (copy)
`=copy`(x, y)
```

```
f_sink(g())
-----------------------   (call-to-sink)
f_sink(g())
```

```
f_sink(notLastReadOf y)
------------------------   (copy-to-sink)
(let tmp = `=dup`(y);
f_sink(tmp))
```

```
f_sink(lastReadOf y)
----------------------   (move-to-sink)
f_sink(y)
`=wasMoved`(y)
```

Object and array construction

Object and array construction is treated as a function call where the function has `sink` parameters.

Destructor removal

`=wasMoved(x)` followed by a `=destroy(x)` operation cancel each other out. An implementation is encouraged to exploit this in order to improve efficiency and code sizes. The current implementation does perform this optimization.

Self assignments

`=sink` in combination with `=wasMoved` can handle self-assignments but it's subtle.

The simple case of `x = x` cannot be turned into `=sink(x, x); =wasMoved(x)` because that would lose `x`'s value. The solution is that simple self-assignments that consist of

- Symbols: `x = x`
- Field access: `x.f = x.f`
- Array, sequence or string access with indices known at compile-time: `x[0] = x[0]`

  are transformed into an empty statement that does nothing. The compiler is free to optimize further cases.

  The complex case looks like a variant of `x = f(x)`, we consider `x = select(rand() < 0.5, x, y)` here:

```
proc select(cond: bool; a, b: sink string): string =
 if cond:
  result = a
 else:
  result = b

proc main =
 var x = "abc"
 var y = "xyz"

 x = select(true, x, y)
```

  Is transformed into:

```
proc select(cond: bool; a, b: sink string): string =
 try:
  if cond:
   `=sink`(result, a)
   `=wasMoved`(a)
  else:
   `=sink`(result, b)
```

```
    `=wasMoved`(b)
  finally:
    `=destroy`(b)
    `=destroy`(a)

proc main =
  var
    x: string
    y: string
  try:
    `=sink`(x, "abc")
    `=sink`(y, "xyz")
    `=sink`(x, select(true,
      let blitTmp = x
      `=wasMoved`(x)
      blitTmp,
      let blitTmp = y
      `=wasMoved`(y)
      blitTmp))
    echo [x]
  finally:
    `=destroy`(y)
    `=destroy`(x)
```

As can be manually verified, this transformation is correct for self-assignments.

Lent type

`proc p(x: sink T)` means that the proc `p` takes ownership of `x`. To eliminate even more creation/copy <-> destruction pairs, a proc's return

type can be annotated as `lent T`. This is useful for "getter" accessors that seek to allow an immutable view into a container.

The `sink` and `lent` annotations allow us to remove most (if not all) superfluous copies and destructions.

`lent T` is like `var T` a hidden pointer. It is proven by the compiler that the pointer does not outlive its origin. No destructor call is injected for expressions of type `lent T` or of type `var T`.

```
type
 Tree = object
  kids: seq[Tree]

proc construct(kids: sink seq[Tree]): Tree =
 result = Tree(kids: kids)

 `=sink`(result.kids, kids); `=wasMoved`(kids)
 `=destroy`(kids)

proc `[]`*(x: Tree; i: int): lent Tree =
 result = x.kids[i]




iterator children*(t: Tree): lent Tree =
 for x in t.kids: yield x

proc main =
```

```
let t = construct(@[construct(@[]), construct(@[])])
echo t[0]
```

The cursor pragma

Under the `--mm:arc|orc` modes Nim's `ref` type is implemented via the same runtime "hooks" and thus via reference counting. This means that cyclic structures cannot be freed immediately (`--mm:orc` ships with a cycle collector). With the `cursor` pragma one can break up cycles declaratively:

```
type
  Node = ref object
    left: Node
    right {.cursor.}: Node
```

But please notice that this is not C++'s weak_ptr, it means the right field is not involved in the reference counting, it is a raw pointer without runtime checks.

Automatic reference counting also has the disadvantage that it introduces overhead when iterating over linked structures. The `cursor` pragma can also be used to avoid this overhead:

```
var it {.cursor.} = listRoot
while it != nil:
  use(it)
  it = it.next
```

In fact, `cursor` more generally prevents object construction/destruction pairs and so can also be useful in other contexts. The alternative solution would be to use raw pointers (`ptr`) instead which is more cumbersome

and also more dangerous for Nim's evolution: Later on, the compiler can try to prove `cursor` pragmas to be safe, but for `ptr` the compiler has to remain silent about possible problems.

Cursor inference / copy elision

The current implementation also performs `cursor` inference. Cursor inference is a form of copy elision.

To see how and when we can do that, think about this question: In `dest = src` when do we really have to *materialize* the full copy? - Only if `dest` or `src` are mutated afterward. If `dest` is a local variable that is simple to analyze. And if `src` is a location derived from a formal parameter, we also know it is not mutated! In other words, we do a compile-time copy-on-write analysis.

This means that "borrowed" views can be written naturally and without explicit pointer indirections:

```
proc main(tab: Table[string, string]) =
 let v = tab["key"]

 use(v)
 useItAgain(v)
```

Hook lifting

The hooks of a tuple type `(A, B, ...)` are generated by lifting the hooks of the involved types `A`, `B`, ... to the tuple type. In other words, a copy `x = y` is implemented as `x[0] = y[0]; x[1] = y[1]; ...`, likewise for `=sink` and `=destroy`.

Other value-based compound types like `object` and `array` are handled

correspondingly. For `object` however, the compiler-generated hooks can be overridden. This can also be important to use an alternative traversal of the involved data structure that is more efficient or in order to avoid deep recursions.

Hook generation

The ability to override a hook leads to a phase ordering problem:

```
type
 Foo[T] = object

proc main =
 var f: Foo[int]



proc `=destroy`[T](f: Foo[T]) =
 discard
```

The solution is to define `proc `=destroy`[T](f: Foo[T])` before it is used. The compiler generates implicit hooks for all types in *strategic places* so that an explicitly provided hook that comes too "late" can be detected reliably. These *strategic places* have been derived from the rewrite rules and are as follows:

- In the construct `let/var x = ...` (var/let binding) hooks are generated for `typeof(x)`.
- In `x = ...` (assignment) hooks are generated for `typeof(x)`.
- In `f(...)` (function call) hooks are generated for `typeof(f(...))`.
- For every sink parameter `x: sink T` the hooks are generated for

`typeof(x)`.

nodestroy pragma

The experimental nodestroy pragma inhibits hook injections. This can be used to specialize the object traversal in order to avoid deep recursions:

```
type Node = ref object
 x, y: int32
 left, right: Node

type Tree = object
 root: Node

proc `=destroy`(t: Tree) {.nodestroy.} =

 var s: seq[Node] = @[t.root]
 while s.len > 0:
  let x = s.pop
  if x.left != nil: s.add(x.left)
  if x.right != nil: s.add(x.right)

  `=dispose`(x)


 `=destroy`(s)
```

As can be seen from the example, this solution is hardly sufficient and should eventually be replaced by a better solution.

Copy on write

String literals are implemented as "copy on write". When assigning a string literal to a variable, a copy of the literal won't be created. Instead the variable simply points to the literal. The literal is shared between different variables which are pointing to it. The copy operation is deferred until the first write.

For example:

```
var x = "abc"
var y = x
y[0] = 'h'
```

The abstraction fails for `addr x` because whether the address is going to be used for mutations is unknown. `prepareMutation` needs to be called before the "address of" operation. For example:

```
var x = "abc"
var y = x

prepareMutation(y)
moveMem(addr y[0], addr x[0], 3)
assert y == "abc"
```