ATS Reference

Chris Double <chris.double@double.co.nz>

Revision History
27 June 2012 CD

Table of Contents

1. Overloaded Functions

The ATS prelude provides some functions that are overloaded with implementations for different types. These can be used instead of the explicit typed functions in many cases to make code easier to understand and more generic.

1.1. print

1.1.1. Definition

```
fun print (p):<!ref> void
```

1.1.2. Description

Prints some representation of *p* to standard output.

1.2. prerr

1.2.1. Definition

```
fun prerr (p):<!ref> void
```

1.2.2. Description

Prints some representation of *p* to standard error.

1.3. Math operators

1.3.1. Definition

```
fun - (a, b):<> c
fun + (a, b):<> c
```

1.3.2. Description

Standard mathematical operators overloaded for different types (pointers, numbers, etc).

1.4. Comparison operators

1.4.1. Definition

```
fun < (a, b):<> bool
fun <= (a, b):<> bool
fun > (a, b):<> bool
fun >= (a, b):<> bool
fun = (a, b):<> bool
```

```
fun <> (a, b):<> bool
fun != (a, b):<> bool
```

1.4.2. Description

Standard comparison operators overloaded for different types (pointers, numbers, etc).

1.5. Pointer operators

1.5.1. Definition

```
fun ~ {l:addr} (p: ptr l):<> bool (l > null)
```

1.5.2. Description

Overloaded for the pointer types to return *true* if the pointer is not null.

1.6. succ

1.6.1. Definition

```
fun succ (a):<> b
```

1.6.2. Description

Returns the successor of a given value. For numbers and pointers this is the value + 1.

1.7. pred

1.7.1. Definition

```
fun predc (a):<> b
```

1.7.2. Description

Returns the predecessor of a given value. For numbers and pointers this is the value - 1.

1.8. padd

1.8.1. Definition

```
fun padd {l:addr} {i:int} (p, i) :<> p
```

1.8.2. Description

Adds *i* to pointer *p*.

1.9. psub

1.9.1. Definition

```
fun psub {l:addr} {i:int} (p, i) :<> p
```

1.9.2. Description

Subtracts *i* from pointer *p*.

1.10. compare

1.10.1. Definition

```
fun compare (a, b):<> Sgn
```

1.10.2. Description

Returns a negative number if *a* is less than *b*, a positive number if *a* is greater than *b* and zero if *a* equals *b*.

1.11. tostring

1.11.1. Definition

```
fun tostring (x):<> strptr1
```

1.11.2. Description

Returns a string representation of value *x*.

2. Linear Lists

list_vt is a polymorphic linear datatype for dealing with linked lists. list_vt.sats contains the definition for datatypes and funtions that operate on linear lists. Use of these routines does not require the garbage collector.

The list_vt datatype has two constructors. list_vt_nil and list_vt_cons. The former represents an empty list and the latter a pair of a viewtype and an existing list.

2.1. Constructors

```
dataviewtype list_vt (a:viewt@ype+, int) =
  | {n:int | n >= 0}
    list_vt_cons (a, n+1) of (a, list_vte (a, n))
  | list_vt_nil (a, 0)
```

The list_vt type is indexed by the type of the item contained in the list and an integer representing the length of the list. A slightly simpler typedef is provided that represents a list of any length:

```
viewtypedef List_vt (a:viewt@ype) = [n:int | n >=0] list_vt (a, n)
```

2.2. list_vt_length_is_nonnegative

2.2.1. Definition

```
prfun list_vt_length_is_nonnegative
  {a:vt0p} {n:int} (xs: !list_vt (a, n)): [n>=0] void
```

2.2.2. Description

Proof function that asserts that the linear list has a non-negative length.

2.3. list_vt_sing

2.3.1. Definition

```
macdef list_vt_sing (x) =
  list_vt_cons (,(x), list_vt_nil ())
```

2.3.2. Description

Creates a *list_vt* containing single element

2.3.3. Example

```
implement main() = {
  val a = list_vt_sing (42)
  val+ ~list_vt_cons (b, ~list_vt_nil ()) = a
  val () = print_int (b)
}
```

2.4. list_vt_pair

2.4.1. Definition

```
macdef list_vt_pair (x1, x2) =
  list_vt_cons (,(x1), list_vt_cons (,(x2), list_vt_nil))
```

2.4.2. Description

Creates a *list_vt* containing two elements

2.4.3. Example

```
implement main() = {
  val a = list_vt_pair ("a", "b")
  val+ ~list_vt_cons (b, c) = a
  val+ ~list_vt_cons (d, ~list_vt_nil ()) = c
  val () = print_string (b)
  val () = print_newline ()
  val () = print_string (d)
```

```
}
```

2.5. list_vt_is_nil

2.5.1. Definition

```
fun{} list_vt_is_nil
  {a:vt0p} {n:int} (xs: !list_vt (a, n)):<> bool (n==0)
```

2.5.2. Description

Returns *true* if the given list is empty.

2.5.3. Example

```
staload _ = "prelude/DATS/list_vt.dats"

implement main() = {
  val a = list_vt_sing (42)
  val+ ~list_vt_cons (b, c) = a
  val () = assertloc (list_vt_is_nil (c))
  val+ ~list_vt_nil () = c
  val () = print_int (b)
}
```

2.6. list_vt_is_cons

2.6.1. Definition

```
fun{} list_vt_is_cons
  {a:vt0p} {n:int} (xs: !list_vt (a, n)):<> bool (n > 0)
```

2.6.2. Description

Returns *true* if the given list is not empty.

2.6.3. Example

```
staload _ = "prelude/DATS/list_vt.dats"

implement main() = {
  val a = list_vt_sing (42)
  val () = assertloc (list_vt_is_cons (a))
  val+ ~list_vt_cons (b, c) = a
  val+ ~list_vt_nil () = c
  val () = print_int (b)
}
```

2.7. list_vt_make_array

2.7.1. Definition

```
fun{a:vt0p}
```

```
list_vt_make_array {n:int}
  (A: &(@[a][n]) >> @[a?!][n], n: size_t n):<> list_vt (a, n)
```

2.7.2. Description

Given a reference to an array, return a linear list of the items in that array.

2.7.3. Example

```
staload _ = "prelude/DATS/list_vt.dats"

implement main() = {
  var !arr = @[string] ("a", "b")
  val a = list_vt_make_array (!arr, 2)
  val+ ~list_vt_cons (b, c) = a
  val+ ~list_vt_cons (d, ~list_vt_nil ()) = c
  val () = print_string (b)
  val () = print_newline ()
  val () = print_string (d)
}
```

2.8. list_vt_of_arraysize

2.8.1. Definition

```
fun{a:vt0p}
list_vt_of_arraysize
  {n:int} (arrsz: arraysize (a, n)):<> list_vt (a, n)
```

2.8.2. Description

Given an arraysize, destructively convert it into a linear list.

2.8.3. Example

```
staload _ = "prelude/DATS/list_vt.dats"

implement main() = {
  val a = list_vt_of_arraysize<int> $arrsz(41, 42)
  val+ ~list_vt_cons (b, c) = a
  val+ ~list_vt_cons (d, ~list_vt_nil ()) = c
  val () = print_int (b)
  val () = print_newline ()
  val () = print_int (d)
}
```

2.9. list_vt_copy

2.9.1. Definition

```
fun{a:t0p}
list_vt_copy {n:int} (xs: !list_vt (a, n)):<> list_vt (a, n)
```

2.9.2. Description

Copy an existing linear list. Note that the items held in the list must be non-linear. The implementation of this template function requires *prelude/DATS/list.dats* to be loaded.

2.9.3. Example

```
staload _ = "prelude/DATS/list_vt.dats"
staload _ = "prelude/DATS/list.dats"

implement main() = {
  val a = list_vt_sing 42
  val b = list_vt_copy (a)
  val+ ~list_vt_cons (c, ~list_vt_nil ()) = a
  val+ ~list_vt_cons (d, ~list_vt_nil ()) = b
  val () = print_int (c)
  val () = print_newline ()
  val () = print_int (d)
}
```

2.10. list_vt_free

2.10.1. Definition

```
fun{a:t0p}
list_vt_free (xs: List_vt a):<> void
```

2.10.2. Description

Frees the linear list, destroying it. The items in the list must be non-linear. See *list_vt_free_fun* for freeing a linear list containing linear resources.

2.10.3. Example

```
staload _ = "prelude/DATS/list_vt.dats"

implement main() = {
  val a = list_vt_of_arraysize<int> $arrsz(41, 42)
  val () = list_vt_free (a)
}
```

2.11. list_vt_free_fun

2.11.1. Definition

```
fun{a:vt0p}
list_vt_free_fun (
  xs: List_vt a, f: (&a >> a?) -<fun> void
) :<> void
```

2.11.2. Description

Frees the linear list, taking a function that will free the items held by the list.

2.11.3. Example

```
staload _ = "prelude/DATS/list_vt.dats"

dataviewtype foo = foo

fn foo_free (f: &foo >> foo?):<> void =
  case+ f of
  | ~foo () => ()

implement main() = {
  val a = list_vt_cons (foo, list_vt_nil ())
  val () = list_vt_free_fun (a, foo_free)
}
```

2.12. list_vt_length

2.12.1. Definition

```
fun{a:vt0p}
list_vt_length {n:int} (xs: !list_vt (a, n)):<> int n
```

2.12.2. Description

Returns the length of the linear list.

2.12.3. Example

```
staload _ = "prelude/DATS/list_vt.dats"

implement main() = {
  val a = list_vt_of_arraysize<int> $arrsz(41, 42)
  val len = list_vt_length (a)
  val () = printf("Length: %d\n", @(len))
  val () = list_vt_free (a)
}
```

2.13. list_vt_make_elt

2.13.1. Definition

```
fun{a:t0p}
list_vt_make_elt {n:nat} (x: a, n: int n):<> list_vt (a, n)
```

2.13.2. Description

Return a linear list composed of length *n* containing *n* copies of *x*. The element type must be non-linear.

2.13.3. Example

```
staload _ ="prelude/DATS/list_vt.dats"

implement main() = {
```

```
  val a = list_vt_make_elt (42, 5)
  val () = printf("List is length: %d\n", @(list_vt_length (a)))
  val () = list_vt_free (a)
}
```

2.14. list_vt_append

2.14.1. Definition

```
fun{a:vt0p}
list_vt_append {m,n:int}
  (xs: list_vt (a, m), ys: list_vt (a, n)):<> list_vt (a, m+n)
```

2.14.2. Description

Returns a linear list containing the elements from *xs* followed by the elements in *ys*. Both *xs* and *ys* are destroyed.

2.14.3. Example

```
staload _ = "prelude/DATS/list_vt.dats"

implement main() = {
  val a = list_vt_of_arraysize<int> $arrsz(41, 42)
  val b = list_vt_of_arraysize<int> $arrsz(43, 44)
  val c = list_vt_append (a, b)
  val () = printf("Length of c = %d\n", @(list_vt_length (c)))
  val () = list_vt_free (c)
}
```

2.15. list_vt_split_at

2.15.1. Definition

```
fun{a:vt0p}
list_vt_split_at {n:int} {i:nat | i <= n}
  (xs: &list_vt (a, n) >> list_vt (a, n-i), i: int i):<> list_vt (a, i)
```

2.15.2. Description

Given a reference to a linear list, and in index into that list *i*, return a list containing the first *i* items and modifies the original list to have the remaining items.

2.15.3. Example

```
staload _ = "prelude/DATS/list_vt.dats"

implement main() = {
  var a = list_vt_of_arraysize<int> $arrsz(41, 42, 43, 44)
  val b = list_vt_split_at (a, 3)

  fun loop {n:nat} (lst: !list_vt (int, n)): void =
    case+ lst of
    | list_vt_nil () => (fold@ lst; print_newline ())
```

```
    | list_vt_cons (i, !rest) => (print_int (i);
                                  loop (!rest);
                                  fold@ lst)

  val () = (print_string ("a:"); print_newline (); loop (a))
  val () = (print_string ("b:"); print_newline (); loop (b))

  val () = list_vt_free (a)
  val () = list_vt_free (b)
}
```

2.16. list_vt_reverse

2.16.1. Definition

```
fun{a:vt0p}
list_vt_reverse {n:int} (xs: list_vt (a, n)):<> list_vt (a, n)
```

2.16.2. Description

Given a linear list, reverses the order of elements and returns the resulting list. The original list is destroyed.

2.16.3. Example

```
staload _ = "prelude/DATS/list_vt.dats"

implement main() = {
  var a = list_vt_of_arraysize<int> $arrsz(41, 42, 43, 44)
  val b = list_vt_reverse (a)

  val () = loop (b) where {
             fun loop {n:nat} (lst: !list_vt (int, n)): void =
               case+ lst of
               | list_vt_nil () => (fold@ lst; print_newline ())
               | list_vt_cons (i, !rest) => (print_int (i);
                                             loop (!rest);
                                             fold@ lst)
           }

  val () = list_vt_free (b)
}
```

2.17. list_vt_reverse_append

2.17.1. Definition

```
fun{a:vt0p}
list_vt_reverse_append {m,n:int}
  (xs: list_vt (a, m), ys: list_vt (a, n)):<> list_vt (a, m+n)
```

2.17.2. Description

Reverses the list *xs* then appends *ys* to the resulting list and returns it.

2.17.3. Example

```
staload _ = "prelude/DATS/list_vt.dats"

implement main() = {
  var a = list_vt_of_arraysize<int> $arrsz(41, 42, 43, 44)
  var b = list_vt_of_arraysize<int> $arrsz(45, 46, 47, 48)

  val c = list_vt_reverse_append (a, b)

  val () = loop (c) where {
             fun loop {n:nat} (lst: !list_vt (int, n)): void =
               case+ lst of
               | list_vt_nil () => (fold@ lst; print_newline ())
               | list_vt_cons (i, !rest) => (printf("%d ", @(i));
                                             loop (!rest);
                                             fold@ lst)
           }

  val () = list_vt_free (c)
}
```

2.18. list_vt_concat

2.18.1. Definition

```
fun{a:vt0p}
list_vt_concat (xss: List_vt (List_vt (a))):<> List_vt (a)
```

2.18.2. Description

Concatenates a list of lists into a single list.

2.18.3. Example

```
staload _ = "prelude/DATS/list_vt.dats"

implement main() = {
  var a = list_vt_of_arraysize<int> $arrsz(1, 2, 3, 4)
  var b = list_vt_of_arraysize<int> $arrsz(5, 6, 7, 8)
  var c = list_vt_of_arraysize<int> $arrsz(9, 10)
  var d = list_vt_cons (a, list_vt_pair (b, c))

  val e = list_vt_concat (d)

  val () = loop (e) where {
             fun loop {n:nat} (lst: !list_vt (int, n)): void =
               case+ lst of
               | list_vt_nil () => (fold@ lst; print_newline ())
               | list_vt_cons (i, !rest) => (printf("%d ", @(i));
                                             loop (!rest);
                                             fold@ lst)
           }

  val () = list_vt_free (e)
}
```

2.19. list_vt_tabulate

2.19.1. Definition

```
fun{a:vt0p}
list_vt_tabulate_funenv
  {v:view} {vt:viewtype} {n:nat} {f:eff}
  (pf: !v | f: (!v | natLt n, !vt) -<f> a, n: int n, env: !vt)
  :<f> list_vt (a, n)

fun{a:vt0p}
list_vt_tabulate_fun {n:nat} {f:eff}
  (f: natLt n -<f> a, n: int n):<f> list_vt (a, n)

fun{a:vt0p}
list_vt_tabulate_vclo {v:view} {n:nat} {f:eff}
  (pf: !v | f: &(!v | natLt n) -<clo,f> a, n: int n):<f> list_vt (a, n)

fun{a:vt0p}
list_vt_tabulate_cloptr {n:nat} {f:eff}
  (f: !(natLt n) -<cloptr,f> a, n: int n):<f> list_vt (a, n)

fun{a:vt0p}
list_vt_tabulate_vcloptr {v:view} {n:nat} {f:eff}
  (pf: !v | f: !(!v | natLt n) -<cloptr,f> a, n: int n):<f> list_vt (a, n)
```

2.19.2. Description

list_vt_tabulate is a family of functions that are used to generate a linear list given a function. The function is called *n* times, with *n* given as an argument. The result of the function is used as the element of the nth position of the generated list.

There is a variant of *list_vt_tabulate* for the different function types available in ATS:

list_vt_tabulate_funenv	General purpose version for any function type
list_vt_tabulate_fun	C type functions (<fun>)
list_vt_tabulate_vclo	Stack allocated closures (<clo>)
list_vt_tabulate_cloptr	Linear closures (<cloptr>)
list_vt_tabulate_vcloptr	Linear closures with a proof argument

2.19.3. Example

```
staload _ = "prelude/DATS/list_vt.dats"

implement main() = {
  val a = list_vt_tabulate_fun<int> (lam (n) =<> n * 2, 4)

  val () = loop (a) where {
             fun loop {n:nat} (lst: list_vt (int, n)): void =
               case+ lst of
               | ~list_vt_nil () => print_newline ()
               | ~list_vt_cons (i, rest) => (printf("%d ", @(i));
                                             loop (rest))
```

```
          }

}
```

2.20. list_vt_foreach

2.20.1. Definition

```
fun{a:vt0p}
list_vt_foreach_funenv
  {v:view} {vt:viewtype} {n:int} {f:eff}
  (pf: !v | xs: !list_vt (a, n), f: !(!v | &a, !vt) -<f> void, env: !vt)
  :<f> void

fun{a:vt0p}
list_vt_foreach_fun {n:int} {f:eff}
  (xs: !list_vt (a, n), f: (&a) -<fun,f> void):<f> void

fun{a:vt0p}
list_vt_foreach_vclo {v:view} {n:int} {f:eff}
  (pf: !v | xs: !list_vt (a, n), f: &(!v | &a) -<clo,f> void):<f> void

fun{a:t0p}
list_vt_foreach_cloptr {n:int} {f:eff}
  (xs: !list_vt (a, n), f: !(&a) -<cloptr,f> void):<f> void

fun{a:t0p}
list_vt_foreach_vcloptr {v:view} {n:int} {f:eff}
  (pf: !v | xs: !list_vt (a, n), f: !(!v | &a) -<cloptr,f> void):<f> void
```

2.20.2. Description

list_vt_foreach is a family of functions that are used to iterate over a list, calling a function for each element in the list. The function receives a reference to the list element as an argument.

There is a variant of *list_vt_foreach* for the different function types available in ATS:

list_vt_foreach_funenv	General purpose version for any function type
list_vt_foreach_fun	C type functions (<fun>)
list_vt_foreach_vclo	Stack allocated closures (<clo>)
list_vt_foreach_cloptr	Linear closures (<cloptr>)
list_vt_foreach_vcloptr	Linear closures with a proof argument

Note that the *cloptr* and *vcloptr* variants work on lists containing types, not viewtypes like the other variants.

2.20.3. Example

```
staload _ = "prelude/DATS/list_vt.dats"

implement main() = {
  val a = list_vt_tabulate_fun<int> (lam (n) =<> n * 2, 4)
  val () = list_vt_foreach_fun<int> (a, lam (x) =<>
```

```
                                        $effmask_all (printf("%d ", @(x))))
  val () = list_vt_free (a)
}
```

2.21. list_vt_iforeach

2.21.1. Definition

```
fun{a:vt0p}
list_vt_iforeach_funenv
  {v:view} {vt:viewtype} {n:int} {f:eff} (
  pf: !v
| xs: !list_vt (a, n), f: (!v | natLt n, &a, !vt) -<fun,f> void, env: !vt
) :<f> void

fun{a:vt0p}
list_vt_iforeach_fun {n:int} {f:eff}
  (xs: !list_vt (a, n), f: (natLt n, &a) -<fun,f> void):<f> void

fun{a:vt0p}
list_vt_iforeach_vclo {v:view} {n:int} {f:eff}
  (pf: !v | xs: !list_vt (a, n), f: &(!v | natLt n, &a) -<clo,f> void):<f> void

fun{a:t0p}
list_vt_iforeach_cloptr {n:int} {f:eff}
  (xs: !list_vt (a, n), f: !(natLt n, &a) -<cloptr,f> void):<f> void

fun{a:t0p}
list_vt_iforeach_vcloptr {v:view} {n:int} {f:eff}
  (pf: !v | xs: !list_vt (a, n), f: !(!v | natLt n, &a) -<cloptr,f> void):<f> void
```

2.21.2. Description

list_vt_iforeach is a family of functions that are used to iterate over a list, calling a function for each element in the list. The function receives as arguments the index of the element in the list and a reference to the list element.

There is a variant of *list_vt_iforeach* for the different function types available in ATS:

list_vt_iforeach_funenv	General purpose version for any function type
list_vt_iforeach_fun	C type functions (<fun>)
list_vt_iforeach_vclo	Stack allocated closures (<clo>)
list_vt_iforeach_cloptr	Linear closures (<cloptr>)
list_vt_iforeach_vcloptr	Linear closures with a proof argument

Note that the *cloptr* and *vcloptr* variants work on lists containing types, not viewtypes like the other variants.

2.21.3. Example

```
staload "prelude/SATS/list_vt.sats"
staload "prelude/DATS/list_vt.dats"
```

```
implement main() = {
  val a = list_vt_tabulate_fun<int> (lam (n) =<> n * 2, 4)
  val () = list_vt_iforeach_fun<int> (a, lam (i, x) =<>
                                        $effmask_all
                                         (printf("%d: %d\n", @(i, x))))
  val () = list_vt_free (a)
}
```

2.22. list_vt_mergesort

2.22.1. Definition

```
fun{a:vt0p}
list_vt_mergesort {n:int}
  (xs: list_vt (a, n), cmp: &(&a, &a) -<clo> int):<> list_vt (a, n)
```

2.22.2. Description

Sorts the list using the merge sort algorithm. Requires a function to provide an ordering for elements in the list.

The comparison function should be stack allocated (has tag <clo>) and requires its arguments to be references. This means you can't use the prelude *compare* function directly. It should return:

negative integer	first argument less than second argument
0	first argument equals second argument
positive integer	first argument is greater than second argument

2.22.3. Example

```
staload _ = "prelude/DATS/list_vt.dats"

implement main() = {
  val a = list_vt_of_arraysize<int> $arrsz (10, 2, 6, 4, 8)
  val () = printf("Unsorted:\n", @())
  val () = list_vt_iforeach_fun<int> (a, lam (i, x) =<>
                                        $effmask_all
                                         (printf("%d: %d\n", @(i, x))))

  val () = printf("Sorted:\n", @())
  var !p_cmp = @lam (a: &int, b: &int): int =<> compare (a, b)
  val b = list_vt_mergesort (a, !p_cmp)
  val () = list_vt_iforeach_fun<int> (b, lam (i, x) =<>
                                        $effmask_all
                                         (printf("%d: %d\n", @(i, x))))
  val () = list_vt_free (b)
```

2.23. list_vt_quicksort

2.23.1. Definition

```
fun{a:vt0p}
list_vt_quicksort {n:int} (xs: !list_vt (a, n), cmp: (&a, &a) -<fun> int):<> void
```

2.23.2. Description

Sorts a list using the quicksort algorithm. Unlike *list_vt_mergesort* this modifies the original list rather than returning the result.

The comparison function should be a standard C function (has tag <fun>) and requires its arguments to be references. This means you can't use the prelude *compare* function directly. It should return:

negative integer	first argument less than second argument
0	first argument equals second argument
positive integer	first argument is greater than second argument

The current *list_vt_quicksort* implementation copies the list into an array and uses the standard C library *qsort* function to sort it. It then copies the data back into the list. Due to the implementation details the following additional files must be loaded to use *list_vt_qucksort*:

- *prelude/DATS/array.dats*
- *libc/SATS/stdlib.sats*

2.23.3. Example

```
staload _ = "libc/SATS/stdlib.sats"
staload _ = "prelude/DATS/list_vt.dats"
staload _ = "prelude/DATS/array.dats"

implement main() = {
  val a = list_vt_of_arraysize<int> $arrsz (10, 2, 6, 4, 8)
  val () = printf("Unsorted:\n", @())
  val () = list_vt_iforeach_fun<int> (a, lam (i, x) =<>
                                        $effmask_all
                                         (printf("%d: %d\n", @(i, x))))

  val () = printf("Sorted:\n", @())
  fn cmp (a: &int, b: &int):<> int = compare (a, b)
  val () = list_vt_quicksort (a, cmp)
  val () = list_vt_iforeach_fun<int> (a, lam (i, x) =<>
                                        $effmask_all
                                         (printf("%d: %d\n", @(i, x))))
  val () = list_vt_free (a)
}
```

3. Pointers

The prelude file *prelude/SATS/pointer.sats* provides functions for manipulating pointers. This includes printing, comparison, pointer arithmetic and getting/setting values stored at the pointer location.

3.1. ptr_is_gtez

3.1.1. Definition

```
praxi ptr_is_gtez
```

```
  {l:addr} (p: ptr l):<> [l >= null] void
```

3.1.2. Description

Proof function asserting that a pointer cannot have a negative address.

3.2. ptr_is_null

3.2.1. Definition

```
fun ptr_is_null (p: ptr):<> bool
```

3.2.2. Description

Given a pointer, return *true* if the pointer is NULL.

3.2.3. Example

```
implement main() = {
  val a = null
  val () = assertloc (ptr_is_null (a))
}
```

3.3. ptr_isnot_null

3.3.1. Definition

```
fun ptr_isnot_null (p: ptr):<> bool
```

3.3.2. Description

Given a pointer, return *true* if the pointer is not NULL.

3.3.3. Example

```
implement main() = {
  val a = null + 1
  val () = assertloc (ptr_isnot_null (a))
}
```

3.4. add_ptr

3.4.1. Definition

```
fun add_ptr_int
  (p: ptr, i: int):<> ptr
overload + with add_ptr_int

fun add_ptr_size
  (p: ptr, sz: size_t):<> ptr
```

```
overload + with add_ptr_size
```

3.4.2. Description

add_ptr is a family of functions for performing pointer arithmetic on the pointer address. The two variants, suffixed by *int* and *size*, allow adding an *int* or a *size_t* respectively. The standard + mathematical operator is overloaded for these functions so pointer arithmetic can be done with normal math operators.

3.4.3. Example

```
implement main() = {
  val a = null
  val () = print (a)
  val () = print_newline ()
  val b = a + 4
  val () = print (b)
  val () = print_newline ()
  val c = add_ptr_int (b, 4)
  val () = print (c)
  val () = print_newline ()
}
```

3.5. sub_ptr

3.5.1. Definition

```
fun sub_ptr_int
  (p: ptr, i: int):<> ptr
overload - with sub_ptr_int

fun sub_ptr_size
  (p: ptr, sz: size_t):<> ptr
overload - with sub_ptr_size
```

3.5.2. Description

sub_ptr is a family of functions for performing pointer arithmetic on the pointer address. The two variants, suffixed by *int* and *size*, allow subtracting an *int* or a *size_t* respectively. The standard - mathematical operator is overloaded for these functions so pointer arithmetic can be done with normal math operators.

3.5.3. Example

```
implement main() = {
  val a = null + 8
  val () = print (a)
  val () = print_newline ()
  val b = a - 4
  val () = print (b)
  val () = print_newline ()
  val c = sub_ptr_int (b, 4)
  val () = print (c)
  val () = print_newline ()
```

```
}
```

3.6. Pointer Comparison

3.6.1. Definition

```
fun lt_ptr_ptr (p1: ptr, p2: ptr):<> bool
and lte_ptr_ptr (p1: ptr, p2: ptr):<> bool

overload < with lt_ptr_ptr
overload <= with lte_ptr_ptr

fun gt_ptr_ptr (p1: ptr, p2: ptr):<> bool
and gte_ptr_ptr (p1: ptr, p2: ptr):<> bool

overload > with gt_ptr_ptr
overload >= with gte_ptr_ptr

fun eq_ptr_ptr (p1: ptr, p2: ptr):<> bool
and neq_ptr_ptr (p1: ptr, p2: ptr):<> bool
overload = with eq_ptr_ptr
overload <> with neq_ptr_ptr
overload != with neq_ptr_ptr
```

3.6.2. Description

This family of functions is used to compare pointers for equality, greater than, less than, not equal, etc. The comparison operators are overloaded to work with the comparison functions.

3.7. ptr1_of_ptr

3.7.1. Definition

```
castfn ptr1_of_ptr (p: ptr):<> [l:addr] ptr l
```

3.7.2. Description

Cast a *ptr* to a *ptr l*, which is a dependently typed pointer indexed by its address.

3.7.3. Example

```
implement main() = {
  val a = null + 4
  val b = ptr1_of_ptr (a)
  val () = print (b)
}
```

3.8. null

3.8.1. Definition

```
val null : ptr null
```

3.8.2. Description

null is the definition of the NULL pointer. It is a dependently typed pointer indexed by the *null* address value.

3.8.3. Example

```
implement main() = {
  val a = null
  val () = print (a)
}
```

3.9. ptr1_is_null

3.9.1. Definition

```
fun ptr1_is_null {l:addr}
  (p: ptr l):<> bool (l==null)
```

3.9.2. Description

Given a dependently typed pointer indexed by the address value, return *true* if the pointer is NULL.

3.9.3. Example

```
implement main() = {
  val a = null
  val () = assertloc (ptr1_is_null (a))
}
```

3.10. ptr1_isnot_null

3.10.1. Definition

```
fun ptr1_isnot_null {l:addr}
  (p: ptr l):<> bool (l > null)
overload ~ with ptr1_isnot_null
```

3.10.2. Description

Given a dependently typed pointer indexed by the address value, return *true* if the pointer is not NULL.

3.10.3. Example

```
implement main() = {
  val a = null + 4
  val () = assertloc (ptr1_isnot_null (a))
}
```

3.11. psucc

3.11.1. Definition

```
fun psucc {l:addr} (p: ptr l):<> ptr (l + 1)
overload succ with psucc
```

3.11.2. Description

Returns the successor of a pointer - the pointer value incremented by one.

3.11.3. Example

```
implement main() = {
  val a = null
  val b = psucc (a)
  val () = print (b)
}
```

3.12. ppred

3.12.1. Definition

```
fun ppred {l:addr} (p: ptr l):<> ptr (l - 1)
overload pred with ppred
```

3.12.2. Description

Returns the predecessor of a pointer - the pointer value decremented by one.

3.12.3. Example

```
implement main() = {
  val a = null
  val b = psucc (a)
  val () = print (b)
  val () = print_newline ()
  val c = ppred (b)
  val () = print (c)
}
```

3.13. padd

3.13.1. Definition

```
symintr padd
fun padd_int
  {l:addr}
  {i:int} (
  p: ptr l, i: int i
) :<> ptr (l + i)
```

```
overload + with padd_int
overload padd with padd_int

fun padd_size
  {l:addr}
  {i:int} (
  p: ptr l, i: size_t i
) :<> ptr (l + i)
overload + with padd_size
overload padd with padd_size
```

3.13.2. Description

An overloaded function that adds an *int* or *size_t* to a pointer.

3.14. psub

3.14.1. Definition

```
symintr psub
fun psub_int
  {l:addr}
  {i:int} (
  p: ptr l, i: int i
) :<> ptr (l - i)
overload - with psub_int
overload psub with psub_int

fun psub_size
  {l:addr}
  {i:int} (
  p: ptr l, i: size_t i
) :<> ptr (l - i)
overload - with psub_size
overload psub with psub_size
```

3.14.2. Description

An overloaded function that subtracts an *int* or *size_t* from a pointer.

3.15. pdiff

3.15.1. Definition

```
fun pdiff
  {l1,l2:addr} (
  p1: ptr l1, p2: ptr l2
) :<> ptrdiff_t (l1 - l2)
overload - with pdiff
```

3.15.2. Description

Subtracts *p2* from *p1* and returns the result as a *ptrdiff_t*.

3.16. Indexed Pointer Comparison

3.16.1. Definition

```
fun plt {l1,l2:addr}
  (p1: ptr l1, p2: ptr l2):<> bool (l1 < l2)
and plte {l1,l2:addr}
  (p1: ptr l1, p2: ptr l2):<> bool (l1 <= l2)
overload < with plt
overload <= with plte

fun pgt {l1,l2:addr}
  (p1: ptr l1, p2: ptr l2):<> bool (l1 > l2)
and pgte {l1,l2:addr}
  (p1: ptr l1, p2: ptr l2):<> bool (l1 >= l2)
overload > with pgt
overload >= with pgte

fun peq {l1,l2:addr}
  (p1: ptr l1, p2: ptr l2):<> bool (l1 == l2)
and pneq {l1,l2:addr}
  (p1: ptr l1, p2: ptr l2):<> bool (l1 <> l2)
overload = with peq
overload <> with pneq
overload != with pneq
```

3.16.2. Description

This family of functions is used to compare dependently typed pointers indexed by their address for equality, greater than, less than, not equal, etc. The comparison operators are overloaded to work with the comparison functions.

3.17. compare_ptr_ptr

3.17.1. Definition

```
fun compare_ptr_ptr (p1: ptr, p2: ptr):<> Sgn
overload compare with compare_ptr_ptr
```

3.17.2. Description

Returns a negative number if *p1* is less than *p2*, a positive number if *p1* is greater than *p2* and zero if *p1* equals *p2*.

3.18. fprint_ptr

3.18.1. Definition

```
fun fprint_ptr {m:file_mode}
  (pf: file_mode_lte (m, w) | out: !FILE m, x: ptr):<!exnref> void
overload fprint with fprint_ptr
```

3.18.2. Description

Prints a representation of the pointer value to a file.

3.19. print_ptr

3.19.1. Definition

--- fun print_ptr (p: ptr):<!ref> void overload print with print_ptr ---

3.19.2. Description

Prints a representation of the pointer to standard output.

3.20. prerr_ptr

3.20.1. Definition

```
fun prerr_ptr (p: ptr):<!ref> void
overload prerr with prerr_ptr
```

3.20.2. Description

Prints a representation of the pointer to standard error.

3.21. tostring_ptr

3.21.1. Definition

```
fun tostring_ptr (p: ptr):<> strptr1
overload tostring with tostring_ptr
```

3.21.2. Description

Returns a string representation of the pointer.

3.22. free_gc_t0ype_addr_trans

3.22.1. Definition

```
praxi free_gc_t0ype_addr_trans
  {a1,a2:t@ype | sizeof a1 == sizeof a2} {l:addr}
  (pf_gc: !free_gc_v (a1, l) >> free_gc_v (a2, l)): void
```

3.22.2. Description

TODO: Is the description right?

A proof function asserting that two values of the same size and type, and at the same memory address, can use the same view for freeing the memory.

3.23. ptr_alloc

3.23.1. Definition

```
fun{a:viewt@ype} ptr_alloc ()
  :<> [l:addr | l > null] (free_gc_v (a?, l), a? @ l | ptr l)
```

3.23.2. Description

Template function for allocating memory for a value of type *a*. It returns a non-NULL pointer to the allocated memory, a view that is required to free the memory later and a view for accessing the value stored at that addressed. The value held at the allocated address is uninitialized.

3.23.3. Example

```
staload _ = "prelude/DATS/pointer.dats"

implement main() = {
  val (pf_gc, pf_int | p_int) = ptr_alloc<int> ()
  val () = !p_int := 42
  val () = print (!p_int)
  val () = ptr_free (pf_gc, pf_int | p_int)
}
```

3.24. ptr_alloc_tsz

3.24.1. Definition

```
fun ptr_alloc_tsz
  {a:viewt@ype} (tsz: sizeof_t a)
  :<> [l:addr | l > null] (free_gc_v (a?, l), a? @ l | ptr l)
```

3.24.2. Description

Like *ptr_alloc*, this function is used for allocating memory for a value of type *a*. Unlike *ptr_alloc*, this is not a template function, so it takes the type of the object as a universal variable and the size as an argument (which must match the actual size of the given universal variable).

It returns a non-NULL pointer to the allocated memory, a view that is required to free the memory later and a view for accessing the value stored at that addressed. The value held at the allocated address is uninitialized.

3.24.3. Example

```
implement main() = {
  val (pf_gc, pf_int | p_int) = ptr_alloc_tsz {int} (sizeof<int>)
  val () = !p_int := 42
  val () = print (!p_int)
  val () = ptr_free (pf_gc, pf_int | p_int)
```

```
}
```

3.25. ptr_free

3.25.1. Definition

```
fun ptr_free
  {a:t@ype} {l:addr} (
  pfgc: free_gc_v (a, l), pfat: a @ l | p: ptr l
) :<> void
```

3.25.2. Description

Free's the memory allocated at the address given by the pointer *p*. Requires the *free_gc_v* view that was obtained when allocating the memory originally and a view for providing access to the type *a* at that address.

3.25.3. Example

See the example for *ptr_alloc*.

3.26. NULLABLE

3.26.1. Definition

```
absprop NULLABLE (a: viewt@ype+) // covariant
```

3.26.2. Description

Abstract prop used when dealing with types that can be initialized to zero (eg. pointers set to NULL, etc). See *ptr_zero* for example usage.

3.27. ptr_zero

3.27.1. Definition

```
fun{a:viewt@ype}
ptr_zero (pf: NULLABLE (a) | x: &a? >> a):<> void
```

3.27.2. Description

Given a reference to an uninitialized pointer type this will result in an initialized type set to zero using *memset*. A proof view of *NULLABLE* for the type is required.

3.27.3. Example

```
staload _ = "prelude/DATS/pointer.dats"

implement main() = {
  var p: ptr
```

```
  val () = ptr_zero (nullable () | p) where {
            extern prfun nullable (): NULLABLE ptr
           }
  val () = print (p)
}
```

3.28. ptr_zero_tsz

3.28.1. Definition

```
fun ptr_zero_tsz
  {a:viewt@ype} (
  pf: NULLABLE (a) | x: &a? >> a, tsz: sizeof_t a
) :<> void
```

3.28.2. Description

Non-template version of *ptr_zero*. Takes the type as a universable variable and the size of that type as an argument.

3.28.3. Example

```
typedef foo = @{ a=int, b=ptr }

implement main() = {
  var p: foo
  val () = ptr_zero_tsz {foo} (nullable () | p, sizeof<foo>) where {
            extern prfun nullable (): NULLABLE foo
           }
  val () = print (p.a)
  val () = print_newline ()
  val () = print (p.b)
```

3.29. ptr_get_t

3.29.1. Definition

```
fun{a:t@ype} ptr_get_t {l:addr} (pf: !a @ l | p: ptr l):<> a
```

3.29.2. Description

Returns the value held at the address of the given pointer. Requires a view of *a @ l* proving a type of *a* is held at that address. The following two lines are equivalent:

```
val v = ptr_get_t<int> (pf_int | p_int)
val v = !p_int
```

In the second line the proof of *int @ l* is implicit - it is found in scope, whereas the *ptr_get_t* usage requires passing it explicitly.

3.29.3. Example

```
staload _ = "prelude/DATS/pointer.dats"
```

```
implement main() = {
  val (pf_gc, pf_int | p_int) = ptr_alloc<int> ()
  val () = ptr_set_t<int> (pf_int | p_int, 42)
  val v = ptr_get_t<int>  (pf_int | p_int)
  val () = print (v)
  val () = ptr_free (pf_gc, pf_int | p_int)
}
```

3.30. ptr_set_t

3.30.1. Definition

```
fun{a:t@ype} ptr_set_t {l:addr}
  (pf: !(a?) @ l >> a @ l | p: ptr l, x: a):<> void
```

3.30.2. Description

Sets the value held at the pointer address to *x*. Requires a view of *a @ l* proving a type of *a* is held at that address. The following two lines are equivalent:

```
val () = ptr_set_t<int> (pf_int | p_int, 42)
val () = !p_int := 42
```

In the second line the proof of *int @ l* is implicit - it is found in scope, whereas the *ptr_set_t* usage requires passing it explicitly.

3.30.3. Example

See *ptr_get_t* for an example of usage.

3.31. ptr_move_t

3.31.1. Definition

```
fun{a:t@ype} ptr_move_t {l1,l2:addr}
  (pf1: !a @ l1, pf2: !(a?) @ l2 >> a @ l2 | p1: ptr l1, p2: ptr l2):<> void
```

3.31.2. Description

Copies the value held at pointer location *p1* into *p2*.

3.31.3. Example

```
staload _ = "prelude/DATS/pointer.dats"

implement main() = {
  val (pf_gc, pf_int | p_int) = ptr_alloc<int> ()
  var x: int = 42
  val () = ptr_move_t<int> (view@ x, pf_int | &x, p_int)
  val () = print (!p_int)
  val () = ptr_free (pf_gc, pf_int | p_int)
}
```

3.32. ptr_move_t_tsz

3.32.1. Definition

```
fun ptr_move_t_tsz {a:t@ype} {l1,l2:addr} (
    pf1: !a @ l1, pf2: !(a?) @ l2 >> a @ l2
  | p1: ptr l1, p2: ptr l2, tsz: sizeof_t a
  ) :<> void
```

3.32.2. Description

A non-template version of *ptr_move_t*. The type is passed as a universal variable and the size is an argument.

3.32.3. Example

```
staload _ = "prelude/DATS/pointer.dats"

typedef foo = @{ a= int, b= string }

implement main() = {
  var x: foo = @{ a= 42, b= "Hello World" }
  val (pf_gc, pf_foo | p_foo) = ptr_alloc<foo> ()
  val () = ptr_move_t_tsz {foo} (view@ x, pf_foo | &x, p_foo, sizeof<foo>)
  val () = print (!p_foo.a)
  val () = print_newline ()
  val () = print (!p_foo.b)
  val () = ptr_free (pf_gc, pf_foo | p_foo)
}
```

3.33. ptr_get_vt

3.33.1. Definition

```
fun{a:viewt@ype}
ptr_get_vt {l:addr}
  (pf: !a @ l >> (a?!) @ l | p: ptr l):<> a
```

3.33.2. Description

Same as *ptr_get_t* but for viewtypes (linear types).

3.33.3. Example

```
staload _ = "prelude/DATS/pointer.dats"

viewtypedef foo (l:addr) = @{ a= int, b= strptr l }
viewtypedef foo = [l:addr] foo l

implement main() = {
  val x = @{ a= 42, b= sprintf ("Hello World", @()) }
  val (pf_gc, pf_foo | p_foo) = ptr_alloc<foo> ()
  val () = ptr_set_vt<foo> (pf_foo | p_foo, x)
```

```
  val v = ptr_get_vt<foo> (pf_foo | p_foo)
  val () = print (v.a)
  val () = print_newline ()
  val () = print (v.b)
  val () = strptr_free (v.b)
  val () = ptr_free (pf_gc, pf_foo | p_foo)
}
```

3.34. ptr_set_vt

3.34.1. Definition

```
fun{a:viewt@ype}
ptr_set_vt {l:addr}
  (pf: !(a?) @ l >> a @ l | p: ptr l, x: a):<> void
```

3.34.2. Description

Same as *ptr_set_t* but for viewtypes (linear types).

3.34.3. Example

See *ptr_get_vt* for an example of usage.

3.35. ptr_move_vt

3.35.1. Definition

```
fun{a:viewt@ype}
ptr_move_vt {l1,l2:addr} (
    pf1: !a @ l1 >> (a?) @ l1, pf2: !(a?) @ l2 >> a @ l2
  | p1: ptr l1, p2: ptr l2
  ) :<> void
```

3.35.2. Description

Like *ptr_move_t* but for viewtypes (linear types). Unlike *ptr_move_t* this is an actual move rather than a copy. Because the type is linear, the original value at the pointer address is no longer accessible after the move operation (it becomes a pointer to an uninitialized type).

3.35.3. Example

```
staload _ = "prelude/DATS/pointer.dats"

viewtypedef foo (l:addr) = @{ a= int, b= strptr l }
viewtypedef foo = [l:addr] foo l

implement main() = {
  var foo1: foo = @{ a= 42, b= sprintf ("Hello World", @()) }
  val (pf_gc, pf_foo | p_foo) = ptr_alloc<foo> ()
  val () = ptr_move_vt<foo> (view@ foo1, pf_foo | &foo1, p_foo)
  val () = print (!p_foo.a)
  val () = print_newline ()
```

```
  val () = print (!p_foo.b)
  val () = strptr_free (!p_foo.b)
  val () = ptr_free (pf_gc, pf_foo | p_foo)
}
```

3.36. ptr_move_vt_tsz

3.36.1. Definition

```
fun ptr_move_vt_tsz
  {a:viewt@ype} {l1,l2:addr} (
    pf1: !a @ l1 >> (a?) @ l1, pf2: !(a?) @ l2 >> a @ l2
  | p: ptr l1, p2: ptr l2, tsz: sizeof_t a
  ) :<> void
```

3.36.2. Description

Non-template version of *ptr_move_vt*, where the viewtype is provided as a universal variable and the size is passed explicitly.

4. Unsafe Functions

The functions and definitions in *prelude/SATS/unsafe.sats* allow performing operations that bypass the type system. They provide a way of saying "I know what I'm doing". For example, casting from one type to another, dereferencing a pointer without a proof, etc.

Some usages of the unsafe functions are to cast from different function types to pass to higher order functions that only accept one type, casting from linear strings to persistent strings temporarily, casting from linear types to non-linear types to share non-linear function implementations in the linear implementation, etc.

Unlike other prelude files, the *unsafe.sats* file is not loaded by default and must be explicitly loaded.

4.1. cast

4.1.1. Definition

```
castfn cast {to:t@ype} {from:t@ype} (x: from):<> to
```

4.1.2. Description

Cast from one type to another. The *from* universal variable is often inferred rather than provided.

4.1.3. Example

```
staload "prelude/SATS/unsafe.sats"

implement main() = {
  val a: int = 5
  val () = print_int (a)
  val () = print_uint (cast {uint} (a))
```

```
}
```

4.2. castvwtp1

4.2.1. Definition

```
castfn castvwtp1 {to:t@ype} {from:viewt@ype} (x: !from):<> to
```

4.2.2. Description

Cast from a viewtype to a type. This is often used to cast from a linear string (*strptr*) to a persistent string (*string*), as a means of passing a read-only view of the linear string. This allows functions that use strings to mainly use the *string* type if they don't modify the string and don't hold a pointer to it.

4.2.3. Example

```
staload "prelude/sats/unsafe.sats"

implement main() = {
  val a: strptr1 = sprintf("hello world", @())
  val () = print_string (castvwtp1 {string} (a))
  val () = strptr_free (a)
}
```

4.3. cast2ptr

4.3.1. Definition

```
castfn cast2ptr {a:type} (x: a):<> ptr
```

4.3.2. Description

Cast from any type to a pointer.

4.4. cast2int

4.4.1. Definition

```
castfn cast2int {a:t@ype} (x: a):<> int
```

4.4.2. Description

Cast from any type to an int.

4.5. cast2uint

4.5.1. Definition

```
castfn cast2uint {a:t@ype} (x: a):<> uint
```

4.5.2. Description

Cast from any type to an unsigned int.

4.6. cast2lint

4.6.1. Definition

```
castfn cast2lint {a:t@ype} (x: a):<> lint
```

4.6.2. Description

Cast from any type to a long int.

4.7. cast2ulint

4.7.1. Definition

```
castfn cast2ulint {a:t@ype} (x: a):<> ulint
```

4.7.2. Description

Cast from any type to an unsigned long int.

4.8. cast2size

4.8.1. Definition

```
castfn cast2size {a:t@ype} (x: a):<> size_t
```

4.8.2. Description

Cast from any type to a *size_t*.

4.9. cast2ssize

```
castfn cast2ssize {a:t@ype} (x: a):<> ssize_t
```

4.9.1. Description

Cast from any type to a *ssize_t*.

4.10. ptrget

4.10.1. Definition

```
fun{a:viewt@ype} ptrget (p: ptr):<> a
```

4.10.2. Description

Returns the value stored at the pointer address without requiring a proof that the type is held at the address.

4.10.3. Example

```
staload "prelude/SATS/unsafe.sats"
staload _ = "prelude/DATS/unsafe.dats"

implement main() = {
  var a: int = 0x65666768
  val p = &a
  var ch1 = ptrget<char> (p)
  var ch2 = ptrget<char> (p + 1)
  var ch3 = ptrget<char> (p + 2)
  var ch4 = ptrget<char> (p + 3)
  val () = printf("%c %c %c %c\n", @(ch1, ch2, ch3, ch4))
}
```

4.11. ptrset

4.11.1. Definition

```
fun{a:viewt@ype} ptrset (p: ptr, x: a):<> void
```

4.11.2. Description

Set a value at a pointer address without requiring a proof of the type existing at that address.

4.11.3. Example

```
staload "prelude/SATS/unsafe.sats"
staload _ = "prelude/DATS/unsafe.dats"

implement main() = {
  var a: int = 0
  val p = &a
  var ch1 = ptrset<char> (p, 'a')
  var ch2 = ptrset<char> (p + 1, 'b')
  var ch3 = ptrset<char> (p + 2, 'c')
  var ch4 = ptrset<char> (p + 3, 'd')
  val () = printf ("%d\n", @(a))
}
```

4.12. vtakeout

4.12.1. Definition

```
absview viewout (v:view) // invariant!
prfun vtakeout {v:view} (pf: !v): viewout (v)
prfun viewout_decode
  {v:view} (pf: viewout (v)): (v, v -<lin,prf> void)
```

4.12.2. Description

Unsure what this is for. Documentation for this appreciated.

5. Strings

The following tables lists the main string types and what they're used for:

string	A non linear string that can only be free'd by the GC
String	A non linear string with a length. This is sometimes referred to in functions as a *string1*.
stropt	An optional type with states for non-null and null non linear strings with a length.
Stropt	A typedef for *stropt* with any length.
strbuf	A linear array of bytes where the last byte is null
strptr	A linear string which must be manually free'd.
strptr0	An strptr that can be null.
strptr1	An strptr that cannot be null.
strptrlen	A linear string with a length

A number of types are defined for characters and bytes:

bytes (int)	A *byte* array with a length
b0ytes (int)	An unintialized *byte* array with a length
chars (int)	A *char* array with a length
c0hars (int)	An unintialized *char* array with a length
c1har	A non-null *char*
c1hars (int)	An array of non-null chars with a length