The Lower (L) Layered Programming Language (Draft 1a)

Robert L. Walton[1]

November 12, 2012

[1]This document is dedicated to the memory of Professor Thomas Cheatham of Harvard University.

NOTICE: Design of this language has been discontinued - R L Walton, June 20, 2014.

1 Introduction

This document describes the Lower Layer Programming Language, or L-Language. See the Introduction to the Layered Programming Languages for basic syntax and for an overview of the related Middle Layer M-Language and Higher Layer H-Language.

Here we will give brief overviews of some of the principle issues addressed by the L-Language.

1.1 Code Targeting

The L-Language is intended to be a target language for compilers of higher level languages. As such it is optimized first to be an easy to use and adequate target language, and second to be reasonably easy to compile into efficient assembly language code.

The L-Language is similar to the C programming language but has additional features intended to give more control over compilation and more type safety in the use, but not the declaration, of data. The L-Language depends upon its powerful macro facility to allow users to define type safe data declaration macros.

1.2 Code Computation

The L-Language permits code to be computed by executing programs, and then compiled and run. The computed code can be an entire function, or can be embedded within a function.

A main reason for this is to support scientific and large data base computing in which code specific to a task must be computed and then optimized.

1.3 Data Declaration

The L-Language supports low level declaration of data and fairly high level type safe optimized usage of declared data. The L-Language also supports macros written in the high level H-Language that extends the L-Language, permitting very capable macros to be written by

users to give users high level type-safe data declaration facilities tailored to specific kinds of data.

The thesis here is that a high level data declaration language that is type safe and efficient for all kinds of data is not practical. So instead tools are provided to create data declaration sublanguages that are type safe and efficient for more limited kinds of data.

One way of explaining our approach is to say that we are taking the normal division of programming into two layers, ***system programming*** and ***application programming***, and we are adding a third layer in-between: ***application system programming***. This last kind of programming uses basic and sometimes type unsafe tools to build advanced and type safe tools for a particular application area, such scientific programming, business accounting, programming with particular data structures, and so forth.

2 Memory

We begin with an overview of L-Language memory, and then provide details in the following subsections.

TBD

2.1 Numbers

Numbers are the basic element of L-Language memory. Numbers are sequences of bits, and each number is one of four types, unsigned integer, signed integer, floating point number, and address:

Name	Kind	Length
uns8	unsigned integer	8 bits
uns16	unsigned integer	16 bits
uns32	unsigned integer	32 bits
uns64	unsigned integer	64 bits
uns128	unsigned integer	128 bits

int8	signed integer	8 bits
int16	signed integer	16 bits
int32	signed integer	32 bits
int64	signed integer	64 bits
int128	signed integer	128 bits
float16	IEEE floating point number	16 bits
float32	IEEE floating point number	32 bits
float64	IEEE floating point number	64 bits
float128	IEEE floating point number	128 bits
adr	address	32 bits or 64 bits (see text)
unsadr	unsigned integer	size of address
intadr	signed integer	size of address

The ***length*** of a number is the number of its bits. Numbers can have different lengths: for example, unsigned integers can have lengths of 8, 16, 32, 64, or 128 bits.

Numbers are stored in random access memory (RAM).

An ***unsigned integer*** of length L is a binary integer with L binary digits (***bits***) and range from 0 to $2^L - 1$.

A ***signed integer*** of length L is a two's complement integer of length L and range from -2^{L-1} to $+2^{L-1} - 1$. This represents the integer I in the given range by the unsigned L-bit integer equal to I modulo 2^L.

A ***floating point number*** of length L is a floating point number represented according to the IEEE 754 standard. The sizes of exponents and mantissas for various floating point number sizes is as follows:

Floating Point Number Size	**Exponent Size**	**Mantissa Size**	**Decimal Digits**	**Maximum Decimal Exponent**
16 bits	5 bits	10 bits	3.31	4.51
32 bits	8 bits	23 bits	7.22	38.23
64 bits	11 bits	52 bits	15.95	307.95
128 bits	15 bits	112 bits	34.02	4931.77

An ***address*** (one kind of '***pointer***') holds a RAM byte address. A address is a 32-bit or 64-bit unsigned integer whose size is determined by the target machine. Some of high order

bits may be required to be all 0's or all 1's, depending upon the target machine. The `unsadr` and `intptr` unsigned and signed integer types of the same size as an address are provided for storing indices and offsets.

A reasonable assumption for 64-bit addresses is that only the low order 48-bits of the address are actually used. This assumption can be used to put other information in the high order 16 bits of a 64-bit number containing an address. For example, an address can be embedded in a 64-bit floating point NaN. The L-Language does not depend upon this assumption, but does provide a builtin function that takes as input a 64-bit integer and two small integers, L and S, and returns a 64-bit address containing the byte address equal to the low order L bits of the input integer left shifted by S. The output may have undefined high order bits if the hardware ignores them when using the output to address memory. For example, if the hardware ignores the high order 20 bits, and uses only the low order 44 bits, this function would just copy its input 64-bit integer to its output if $L \geq 44$ and $S = 0$.[2]

Variables can be declared to be of non-address numeric type by declarations of the form

type-name variable-name

For example,

```
int32 i
uns128 u
float64 f
```

Variables can be declared to be of address type by declarations of the form

`adr` *variable-name* `->` *variable-declaration*

For example,

```
adr ip1 -> int32
adr ip2 -> int32 i2
adr ipp3 -> adr ip3 -> int32 i3
```

declare variables `ip1` and `ip2` that store the address of an `int32` variable, and a variable `ipp3` that stores the address of a variable that stores the address of an `int32` variable. The variable pointed at by an address can be given a name; e.g., `i2` is the variable pointed at by

[2]The I86 64-bit architecture uses only the low order 48 bits of an address, but requires the high order 17 bits to all be the same, either all 1's or all 0's. However, as it is unlikely that there will ever be an allocated memory region that includes address 0 in its interior, it makes no significant difference whether we consider addresses to be unsigned or signed.

the address stored in `ip2`. The unary '`*`' operator is used to reference the variable pointed at by an address, so here '`* ip1`', '`* ip2`', '`i2`', '`** ipp3`', '`* ip3`', and '`i3`' reference an `int32` integer variable, while '`ip1`', '`ip2`' '`* ipp3`', and '`ip3`', reference an `adr` variable that stores the address of an `int32` variable.

Some of the names in address type variable declarations may be omitted, as in

```
adr ip -> int32
adr    -> int32 i
```

In both these cases the value stored in the variable is the address of a 32-bit integer. However, `ip` names the address and `i` names the integer. The unary '`&`' operator is the inverse of '`*`', so '`* ip`' can be used to name the integer, and '`& i`' can be used to name the address. Either of these names can be used to assign values, as in

```
* ip = 5
& i = ip
```

Variable declarations within a code block (2.5^{p15}) are actually short forms for the declaration of constant stack addresses. For example, within a function code block the variable declarations

```
int32 i
stack adr ip -> int32
```

are equivalent to

```
constant stack adr = some-stack-address -> int32 i
constant stack adr = some-stack-address -> stack adr ip -> int32
```

so that '`ip = & i`' is a legal assignment. The compiler arranges for appropriate stack addresses to be computed. '`& i`' and '`& ip`' are both '`constant`' and '`stack`', meaning respectively that they cannot be changed and that they have a lifetime guaranteed to last only as long as function code block is executing.

Note that if '`stack adr ip`' were changed to '`adr ip`' the assignment '`ip = & i`' would no longer be legal, because the lifetime of '`& i`' is shorter than the indefinite lifetime promised by '`adr ip`' unqualified by '`stack`'.

2.2 Subtypes

A subtype of a parent type is a type whose values are formatted and aligned in the same manner as values of the parent type and whose values can be implicitly converted to or from

the parent type by unchecked code. Given this, it is possible to define checked conversion functions.

One use of a subtype is to make a promise that a value is in a given range. The following is an example of a subtype that has the range `0 .. max index-1`:

```
define type index int = subtype of int32

external:
    constant int32 max index

inline index int .convert. ( int32 i ) unchecked:
    assert ( 0 <= i && i < max index )
    return i

inline int32 .convert. ( index int i ) unchecked:
    int32 j = i
    fact ( 0 <= j && j < max index )
    return j

float64 x[max index]
float64 getx ( index int i ) = x[i]
```

Here '`x[i]`' will expand to code containing an implicit conversion of `i` from `index int` to `int32` and also a check that `i` is in the proper range to be in index of `x`, so the code expansion will effectively contain:

```
    fact ( 0 <= i && i < max index )
    assert ( 0 <= i && i < max index )
```

The '`fact`' statement allows the '`assert`' statement to be optimized away.

A second use of a subtype is to reserve values of the parent type for special meaning. For example, an `adr` address value in the range from 0 through 4095 might be interpreted as an integer and not an address, under the assumption that low addresses are not used by the memory system. An example of a subtype implementing this is:

```
define type special adr = subtype of adr
    // A special adr is either an adr or an integer
    // in the range [0,4096).

inline bool is int ( special adr a ) unchecked:
```

```
    unsadr x = a
    return x < 4096

inline bool is adr ( special adr a ) unchecked:
    unsadr x = a
    return x >= 4096

inline adr .convert. ( special adr a ) unchecked:
    unsadr x = a
    assert ( x >= 4096 )
    return a

inline unsadr .convert. ( special adr a ) unchecked:
    unsadr x = a
    assert ( x < 4096 )
    return x

inline special adr .convert. ( unsadr x ) unchecked:
    assert ( x < 4096 )
    adr a = x
    return a

inline special adr .convert. ( adr a ) unchecked:
    unsadr x = a
    assert ( x >= 4096 )
        // We could omit this assert if we knew that
        // every adr was >= 4096.
    return a

float64 x
stack special adr y = & x -> float64
stack special adr z = 23 -> float64
assert ( ! is int ( y ) )
assert ( is int ( z ) )
float64 x2 = * y   // OK
float64 z2 = * z   // Causes assert violation error!
```

2.3 Random Access Memory (RAM) and Blocks

RAM is a set of address/byte pairs. Each 8-bit byte of RAM has an ***address*** that is an unsigned integer. It is possible for two addresses to refer to the same byte, or for an address to refer to no byte. An address that refers to a byte is said to be ***allocated***, and an address that refers to no byte is said to be ***deallocated***. An allocated byte that has two (or more) distinct addresses is said to be ***shared***, and a byte with exactly one address is ***unshared***. A deallocated byte is neither shared nor unshared.

A ***RAM block*** is a sequence of RAM bytes with consecutively increasing addresses. The ***origin*** of the block is the first address, and the ***length*** of the block is the number of bytes.

A RAM block is ***allocated*** if the addresses of all its bytes are allocated, and is ***deallocated*** if the addresses of all of its bytes are deallocated. A RAM block is ***unshared*** if all its bytes are unshared. A RAM block is ***shared*** if it is one of a set of several blocks (sequences of consecutive addresses) such that the n'th byte of each block in the set is the same. Note that a RAM block can be neither allocated nor deallocated, or neither shared nor unshared; e.g., some block bytes may be allocated and some deallocated. Also note that a shared byte must have two distinct addresses, and this has nothing to do with whether the byte is in two distinct blocks that overlap.

The address space is divided into ***pages***, which are RAM blocks that have an implementation determined length that is a power of two, e.g., 4096 byte pages, and an address that is an exact multiple of this length. There are L-Language operations which call the operating system to map an address page to physical memory, thus allocating a page of RAM. There are operations to deallocate a page, and to make two pages be shared (map to the same bytes of RAM).

Pages are the units of allocation and sharing. Each page of the address space is either allocated or deallocated, and each allocated page is either shared or unshared. The only way for an arbitrary RAM block to have allocated bytes is to overlap an allocated page, and similarly for deallocated bytes, shared bytes, and unshared bytes.

A ***segment*** is a contiguous sequence of pages all of which have the same allocation and sharing status.

The builtin RAM blocks in L-Language are link blocks, frame blocks (p15), code RAM blocks(p15), numbers, and segments. All non-segment blocks are contained within segments. User defined (i.e., non-builtin) blocks can be contained within segments.

Certain blocks are '***independent blocks***'. These have the property that no two distinct independent blocks can overlap. Link blocks and frame blocks are independent. Blocks

allocated by user provided heap managment routines are also typically independent.

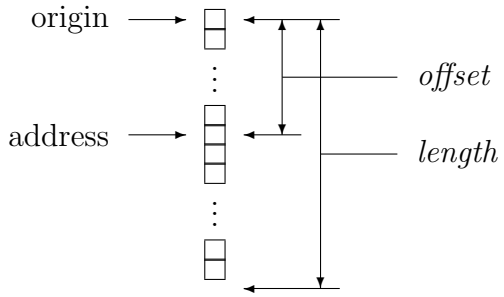
User defined block data types may be declared by declarations of the form

define type *type-name* `=` *type-flag** `(` *length*`,` *alignment*`,` *offset* `)`

type-flag	::=	`floating` | `address` | `extendable`
	|	`locatable variable` | `locatable value` | `descriptor`
length	::=	*natural-number*
alignment	::=	*natural-number*
offset	::=	*integer*

The *type-flags*, *length*, *alignment*, and *offset* are just the information needed to allocate a variable of the type to memory and control and optimize copying of the values of the type as call arguments and return values.

The layout of a block of the defined type in memory is:



address modulo *alignment* = 0

The ***address*** of the block is offset from the origin of the block by the ***offset*** of block. The address must be an exact multiple of the ***alignment*** of the block. The length of the block is an unsigned integer. The alignment is an unsigned integer equal to a power of two that defaults to `1`. The offset is a signed integer that defaults to `0`.

A block can actually have bytes both before and after those described by its type. Thus the type specifies that every allocated block of the type will contain allocated bytes at the addresses

address - offset
address - offset + 1
address - offset + 2
. . . . . .
address - offset + length - 1

These are the ***required bytes*** of the block. In addition the block may contain ***optional bytes*** either before or after the required bytes.

Note that if the *offset* is negative, the block address does not address a required byte, and may not address any block byte. Similarly if the *offset* is equal to or greater than the block *length*.

The *type-flags* in a type definition modify the way that blocks are passed between a function caller and the function execution. The possible flags are:

floating Specifies the value is an IEEE floating point number that can be moved through floating point registers in standard hardware without harm. Note that some Not-a-Number (NaN) and denormalized IEEE values may be modified by this process.

address Specifies the value is a standard machine address (such as type `adr`) and may be moved through address registers in standard hardware without harm. On some hardware, for example, addresses must be in a certain range. For example, on current (2012) I84_64 hardware an address is a 64 bit integer the top 17 bits of which must be identical.

extendable Specifies the value may contain optional bytes, as indicated above. Such values cannot be copied between a caller and a called function execution, but addresses pointing at them may be copied instead.

descriptor Specifies there is only one possible value of the type, and any variable of that value will be initialized to that value. This value is necessarily determined by the type, and in general describes the type: see 2.6^{p17}.

locatable variable Specifies the value must be stored in the thread stack in a locatable variable: see TBD for details. Such values cannot be moved from their locatable variable, and so cannot be copied between a caller and a called function execution, but addresses pointing at them may be copied instead.

locatable value Specifies the value must be stored in the stack in some locatable variable, though if it so stored, it may also be stored elsewhere: see TBD for details. Such values can be copied between a caller and a called function execution. When copied from caller to called function, the called function may assume the value is stored in a locatable

variable within the caller frame. But when returned from a called function to a caller, the value must be copied into locatable variable before any interrupt occurs.

Some of the standard default type definitions are

```
define type int32 = (4,0,0)
define type float64 = floating (8,0,0)
// On some hardware:
    define type adr = address (8,0,0)
    define type unsadr = (8,0,0)
    define type insadr = (8,0,0)
// On other hardware:
    define type adr = address (4,0,0)
    define type unsadr = (4,0,0)
    define type insadr = (4,0,0)
```

Blocks contain numbers (including direct addresses) and subblocks. Access to these is provided by displacement values. For example,

```
define type type1 = (48,8)
unchecked:
    constant disp member1 = 0 -> uns16 @ type1
    constant disp member2 = 2 -> uns16 @ type1
    constant disp member3 = 4 -> uns32 @ type1
    constant disp member4 = 8 -> float32 @ type1
    constant disp member5 = 12 -> adr @ type1 -> type1
. . . . .
type1 x
x.member1 = 9
x.member2 = -19
x.member3 = 190
x.member4 = -0.01 * x.member3
x.member5 = & x
```

A type of the form

disp `->` *type1* `@` *type2*

denotes a displacement of a block of type *type1* inside a block of type *type2*. The displacement itself is a signed integer with the same size and characteristics as an `intadr` signed integer.

A statement of the form

disp member-name = *value* -> *type1* @ *type2*

assigns the *value* to the *member-name* variable that has the above type.

Above '`unchecked:`' introduces a block of unchecked code, that is, code in which normal type checking rules do not apply. Using = to assign a signed integer to a displacement variable is an unchecked operation that can be used only in unchecked code.

The type of the expression '`x.member1`' is

```
constant stack adr -> uns16 V @ type1.member1
```

and not

```
constant stack adr V -> uns16 @ type1.member1
```

In these types the *variable-name* `V` is used to indicate which value, the integer or the address, is the value of the expression. This is the value that will be read if the expression appears to the right of = and written if the expression appears to the left. This value is also called the ***value position*** inside the address type, and we say that `V` specifies the value position inside the address type. Any other *variable-name* besides `V` could be used to specify the value position in an address type.

Therefore the statement

```
x.member1 = 9
```

sets an `uns16` integer variable, and not an `adr` constant.

Notice that the expression '`x.member1`' has type

```
constant stack adr -> uns16 V @ type1.member1
```

and not

```
constant stack adr -> uns16 V
```

The difference between the type '`uns16 @ type1.member1`' and the type '`uns16`' is that the former specifies an `uns16` value that is inside a `type1` value, and is in fact at offset `member1` inside that `type1` value. For targets of `adr` data this extra information is used by the compiler to determine when variables can be aliased. For targets of `disp` data, as in

```
constant disp member1 = 0 -> uns16 @ type1
```

this extra information is used by the compiler to type check expressions such as '`x.member1`'.

2.4 Independent Block Types

Some RAM blocks are '***independent***'. Two different independent blocks are guarenteed to not overlap each other.

There are three basic kinds of independent blocks: static blocks, stack blocks, and heap blocks.

Static blocks are never deallocated or relocated. They can be ***link blocks*** that are allocated when a program is loaded, or they can be user allocated blocks that are never deallocated.

Stack blocks are allocated within thread stack frames (see 2.5^{p15}). A stack block is allocated by a code block execution, and is deallocated when that code block execution terminates.

Heap blocks are user allocated blocks that may be deallocated. Usually they are garbage collectible.

All the heap blocks that are garbage collectible by the same garbage collection algorithm are grouped together into a '***heap***'. For each heap there is a list of static block locations that might hold pointers into blocks in the heap. And for each heap and each thread stack there is a separate list of stack block locations that might hold pointers into blocks in the heap.

Independent RAM blocks may be relocated (i.e., moved) in RAM. This applies even to link blocks.

2.5 Code and Frame Blocks

A ***code block*** is a sequence of executable program statements and code subblocks. Some code blocks are subblocks nested inside other code blocks, and some are not.

Code blocks are sequences of lexemes, and are not RAM blocks. Associated with each code block is a ***code RAM block*** which is the code block translated into machine code. The code RAM block of a code subblock need not be part of the code RAM block of the containing code block, but must be associated with the latter. Code RAM blocks may be static, stack, or heap blocks.

A function is a code block whose execution is initiated by a call statement. The execution of the code block containing the call statement is interrupted in order to execute the function code block. When the call returns, the interrupted execution resumes.

Similarly the execution of a code subblock interrupts the execution of its containing code block.

Code block executions are grouped into ***threads***. At any time, only one code block may be executing statements within a given thread.

Each thread has a ***stack***. When a code block begins executing, a ***frame*** is allocated to the stack of the thread containing the code block execution. This frame is a RAM block that holds data for the code block execution. Thus there is a 1-1 correspondence between code block executions and frame blocks.

A thread stack is a sequence of RAM blocks. A frame has a fixed part which is allocated when its code block execution begins and which has a size determined at compile time. A frame may also have extensions which are allocated during its code block execution. Deallocation of all parts of the frame occurs precisely when the frame's code block execution terminates.

Frame blocks do not overlap each other, static blocks, or heap blocks. Therefore blocks allocated within frame blocks that do not overlap other blocks within the same frame are independent. Stack variables are independent blocks allocated to a frame or its extensions. A variable whose size is determined at compile time and which is sufficiently small will be allocated in the fixed part of the frame. For example, the declaration

```
int64 x;
```

allocates an independent block for `x` in the frame associated with the execution of a code block containing this declaration, and the complete type of `x` is

```
constant stack adr -> int64 x;
```

where `& x` is the address of `x` within the fixed part of the frame. However a variable of large size, or a variable of size not known at compile time, will be allocated to an extension of the frame. For example, the declarations

```
define type my type(uns32 M) = (24+8*M,0,8)
.  .  .  .  .
constant uns32 n = ...
my type(n) y
```

where `M` is a type function argument and `n` an argument value causes `y` to be allocated as an independent block in an extension of the frame, and the complete type of `y` is

```
constant stack adr -> constant stack adr -> my type(n) y;
```

where `& & y` is the address of the variable `& y` that is allocated within the fixed part of the frame, and the value of this variable is the address of `y`.

2.6 Names, Types, Variables, and Functions

A '***name***' is used to name a variable, function call, type, or qualifier. All names are sequences of words, numbers, quoted strings, and bracketted lists of arguments.

In certain contexts several names may appear in a row. The name of a variable or function call may be immediately preceded by the name of a type or by qualifier names, and the name of a type may be immediately preceded by qualifier names. In these contexts qualifier names and type names are recognized because they have been declared in advance.

Names may be either '***complete***' or '***prototype***'. In a complete name arguments are given as values. In a prototype name, the arguments are represented by argument type/name pairs (with optional qualifiers).

Names may be printed and matched with each other. Matching may induce assignment of argument names in prototypes to values supplied by complete names. A prototype name may be matched with a complete name but not with another prototype name.

For example,

> `sin ( float64 x )` matched with `sin ( 3.141529 )`
> assigns `x = 3.141529`
>
> if `x` is of type `int32` then
> `max ( type T, T v, T w )` matched to `max ( x , 2000 )`
> assigns `T = int32`, `v = x`, and `w = 2000`.

Matching a prototype name to a complete name is complex and allows omitted arguments, most particularly type arguments such as `T` in the example, to be deduced via match generated assignments. Implicit conversions may also be allowed, such as the conversion of `2000` to type `T` in the example. On the other hand, matching two complete names merely involves checking for equality of names after fully evaluating arguments.

Names have the syntax:

name	::=	*named-argument-list named-argument-list**
	|	*named-argument-list** *simple-name*
simple-name	::=	*word* { *word* | *number* }*
	|	*quoted-string*
named-argument-list	::=	*simple-name argument-list argument-list**
argument-list	::=	`(` *argument* { `,` *argument* }* `)`
	|	`[` *argument* { `,` *argument* }* `]`
	|	ditto but with brackets other than `( )` or `[ ]` and/or separators other than `,`
argument	::=	*expression* | *variable-declaration*

Types are declared by ***type declarations*** which have the syntax:

type-declaration	::=	`define type` *qualifier** *type-name*
	|	`define type` *qualifier** *type-name* `=` *type-flag** *type-spec*
type-name	::=	*name*
type-spec	::=	*block-type-spec* | *subtype-type-spec*
block-type-spec	::=	`(` *length*`,` *alignment*`,` *offset* `)`
	|	`(` *length*`,` *alignment* `)`
	|	`(` *length* `)`
subtype-type-spec	::=	`subtype of` *type-name*
type-flag	::=	`floating` | `address` | `extendable`
	|	`locatable variable` | `locatable value` | `descriptor`

Variables are declared by ***variable declarations*** which have the syntax:

variable-declaration	::=	*simple-variable-declaration* { `->` *simple-variable-declaration* }*
simple-variable-declaration	::=	*qualifier** *type-name container**
	|	*qualifier** *type-name container** *variable-name*
	|	*qualifier** *type-name container** *variable-name* `=` *initializer*
	|	*qualifier** *type-name container** *variable-name* `==` *equivalent*
variable-name	::=	*name*
container	::=	`@` *qualifier** *type-name*

Functions are declared by ***function declarations*** which have the syntax:

function-declaration ::= *function-output-spec qualifier⋆ function-name*
function-output-spec ::= *qualifier⋆ type-name*
| *argument-list*
function-name ::= *name*

Qualifier declarations are described later (2.14[p39]). ***Qualifiers*** have the syntax:

qualifier-name ::= *name*

2.7 Descriptors

A ***descriptor*** is a run-time value that gives information about a type. A descriptor has a type that has the '**descriptor**' *type-flag*, and every such type has the special property that there is only one value possible of that type[3].

The '**basic**' descriptor includes the length, alignment, offset, and flags of a type. An example use is

```
basic(int32) b;
unsadr s    = b.length       // Length of int32 (4)
unsadr a    = b.alignment    // Alignment of int32 (4)
intadr o    = b.offset       // Offset of int32 (0)
type flags f = b.flags       // Flags of int32 (none)
```

If T is a type, then `basic(` T `)` is a descriptor type that has a unique value computable by the compiler and loader, and this value has members that give the basic parameters of the type T, namely T's length, alignment, offset, and flags.

It is possible to create new descriptors by defining a descriptor type and providing for it a constructor that can be run to compute the one and only value of that type. This constructor can only take omitted arguments, which are generally provided by the type itself. If the descriptor type is defined as a subtype, a conversion function also needs to be supplied. For example:

```
define type constant length ( type T ) = descriptor subtype of unsadr
( constant length(T) s) .construct. ( type T ) unchecked:
    basic(T) b
    s = b.length
( unsadr r ) .convert. ( type T, length(T) s ) unchecked:
    r = s
```

[3]This is a central idea in the Haskell programming language

Values of descriptor types must be declared to be '`constant`' (see p45). Constructors of such values may not be invoked with actual arguments; that is, all their arguments must be omitted (and infered from the type of the output).

2.8 Traceable Types

The traceable type flags specify that values of a given type are locatable if they are stored in '`static`' or '`stack`' locations, that is, in locations with an address of '`adr`' type that has either the '`static`' or '`stack`' qualifier, so the location is either static or in a code block frame.

traceable location A stack or static location of a '`traceable location`' type can be found by appropriate trace operations.

Note that traceable location types cannot be used for arguments or return values of functions. However, addresses of locations of these types can be used.

Traceable location types are typically used for open files, exception catch data, etc.

traceable value If a value of a '`traceable value`' type is stored in a stack or static location, then at least one such location containing the value can be found by appropriate trace operations. If one tracable location contains the value, the value may be stored in other untraceable stack or static locations.

Traceable value types can be used for arguments and return values of functions. When used as an argument, traceable values are stored in a traceable location by the caller before the call is made. When used as a return value, traceable return values which are to be stored in a stack or static location are stored in a caller traceable location as part of the call return operation.

Traceable value types are typically used for data of interest to a garbage collector.

Note that the traceable type flags are not allowed for types of location that are not stack or static.

2.9 Pointer Types

The standard pointer type is `adr`, which denotes an address. But any type can be used as a pointer type in variable declarations.

The general syntax of declarations in involving pointer types is

pointer-declaration	::=	*pointer-variable-declaration*
		`->` *target-variable-declaration*
pointer-variable-declaration	::=	*simple-variable-declaration*
target-variable-declaration	::=	*pointer-variable-declaration*
	|	*simple-variable-declaration*
simple-variable-declaration	::=	see p18

A *pointer-declaration* can be used specify a ***pointer type***. When this is done, a single arbitrary variable name is placed within the declaration to indicate the '***value position***' of the type. Thus in the pointer types

```
adr V -> int32
adr -> int32 W
```

the first is the type of an address pointer variable pointing at an integer target variable, and the second is the type of an integer target variable pointed at by an address variable. The variable names choosen, in this case `V` and `W`, have no significance; any variable names could be substituted here.

The operators '`*`' and '`&`' are automatically defined for pointer types and variables of these types. These operators merely move the value position in the types. When applied to variables, these operators permit reference to pointer and target variables not explicitly named. Thus

`*`	converts type	*type1* `V ->` *type2*	to	*type1* `->` *type2* `V`
`&`	converts type	*type1* `->` *type2* `V`	to	*type1* `V ->` *type2*

and in

```
int32 z                 // Implicitly: constant stack adr -> int32 z
stack adr y -> int32    // Implicitly: constant stack adr ->
                        //                 stack adr y -> int32
y = & z                 // & z is address of z
int32 w = * y           // * y is target of y
```

Whether something is a legitimate pointer type depends upon whether it may be used to read or write values. Reading is done by the implicitly invoked '`.get.`' function, and writing by the implicitly invoked '`.set.`' function.

For example:

```
int8 x -> float64 y
float64 z = y     // Translates to .set.(&z,.get.(.get.(&x)));
```

```
                    // or to simplify z = .get.(x) or z = .get.(&y);
                    // .get.(x) is probably undefined as x is int8.
y = z               // Translates to .set.(.get(&x),.get.(&z));
                    // or to simplify .set.(x,z);
                    // .set.(x,...) is probably undefined.
```

If `x` has a type of the form '*T*1`->`*T*2 `x`', then '`x = ...`' translates to '`.set.(&x,...)`' and '`... = ... x ...`' translates to '`... = ... .get.(&x) ...`', but we simplify by writing the untranslated versions in these cases.

The `.get.` and `.set.` functions are likely to be undefined for `int8` pointers. But in special cases it may be convenient to define them. For example, if `int8` values are used as file descriptors to reference open files, one might have:

```
define type open file = ...
open file .get. ( int8 V -> open file )

( int8 V -> open file ) open ( ... )

int8 x -> open file
x = open ( ... )
open file z = * x  // Translates to z = .get.(x)
```

Although this is technically correct, it does not make much sense for an '`open file`' to be passed directly as a value. Rather its address should be passed. So if `int8` values are to be used as file descriptors, they should be convertable to `adr` values, as in the following improved example:

```
define type open file = ...
( adr V -> open file ) .convert. ( int8 V -> open file )
    // Declares conversion function which is not defined here.
unchecked:
    constant disp state = 0 -> uns16 @ open file
    . . . . .

int8 V -> open file open ( ... )
    // Declares function that opens a file.

int8 x -> open file
x = open ( ... )
```

```
uns16 y = x.state    // Translates to
                     // y = .get.((.convert.(x)).state))
```

Here the implicit `.convert.` function is invoked automatically because X`.state` is only defined if X has type '`adr -> open file`'.

We end this section with a more refined example. Suppose '`ptr`' is an indirect address, that is, the address of the real address of the variable pointed at. Further suppose the real address has the user defined '`relocatable`' lifetime qualifier, as described in Section 2.14.1[p42], and the location where the real address is stored has the '`static`' lifetime qualifier. This could all be implemented by the code:

```
define type ptr = subtype of adr

inline relocatable adr .convert. ( ptr p ) unchecked:
    adr p2 = p -> relocatable adr; return * p2
inline ptr .convert. ( static adr p -> relocatable adr ) unchecked:
    return p

static:
    relocatable adr stub -> uns32
    // The full type of stub is:
    //     constant static adr -> relocatable adr stub -> uns32

ptr p = & stub -> uns32  // Translates to p = .convert.(&stub)
* p = 5                  // Translates to .set(.convert.(p),5)
uns32 y = * p            // Translates to y = .get.(.convert.(p))
ptr q = & stub -> uns64  // Error! No implicit conversion available.
```

A pointer value of type '$T1$`->`$T2$' can be converted to type '$T3$`->`$T4$' by a '`.convert.`' implicit conversion function that converts any value of type $T1$ to a value of type $T3$, but only if $T2$ is identical to $T4$. Thus the assignment of `p` above is legal and the assignment of `q` is not.

2.10 Displacements

Displacement types are just like pointer types in so far as declaration and assignment is concerned, but differ in not having `.get` and `.set` functions, but instead having the `"."` operator that combines an address and a displacement.

Builtin "." operators combine `disp` and `adr` values. Specifically, given values of types

`adr A ->` $T1$
`disp D ->` $T2$ `@` $T1$

the expression `A.D` adds the value of `D` to the value of `A` to produce a value of type

`adr AD ->` $T2$ `@` $T1$

2.11 Tuples and Parameter Memory

A ***tuple*** is a sequence of zero or more values, where the values need not all have the same type. Tuples are surrounded by brackets and their values are separated by separators. The kind of bracket and separator determine the '***type***' of the tuple. For example, there can be '`(,)`'-tuples that are values bracketed with parentheses '`( )`' and separated by commas '`,`', and there can be '`[;]`'-tuples that are values bracketed with square brackets '`[ ]`' and separated by semi-colons '`;`'.

A tuple is not a first class datum; it cannot be stored in most kinds of memory. Tuples are temporary data used to pass values to functions and return values from functions. A call to a function typically involves two tuples, one holding the ***arguments*** of the function and another holding the ***return values*** of the function. However calls may involve many tuples or no tuples.

It is also possible for one code block to transfer to another. This is like a call in which the first code block execution terminates just as the second code block execution begins. A tuple can be transferred along with control.

Nested tuples are ***flattened*** when they have the same tuple type, that is, the same brackets and separators. For example, '`(1, 2, (3, 4), 5)`' is equivalent to '`(1, 2, 3, 4, 5)`'. As another example, if the function call '`function1()`' returns the tuple '`(1, 2, 3)`', then the expression '`(function1())`' is equivalent to '`((1, 2, 3))`' which is equivalent to '`(1, 2, 3)`'. Note that a singleton tuple has no separator, but for purposes of tuple type matching 'no separator' matches every specific separator.

Tuples are stored in ***parameter memory***, which is a kind of memory used to pass function arguments and results. It is implementation dependent whether parts of parameter memory are actually RAM memory; they may be registers, or some other kind of memory. Different tuple values are independent blocks; they do not overlap other tuple values or blocks not stored in parameter memory.

2.12 Clusters

A ***cluster*** is a group of related variables. One variable of the cluster is the ***base variable*** of the cluster, and the other variables have names that are derived from the name of the base variable using the syntax:

variable-name ::= *base-name member-selector**

member-selector ::= `.`*member-name*
| `[`*member-index-list*`]`

member-index-list ::= *member-index*
| *member-index* `,` *member-index-list*

member-index ::= *integer-constant-expression*

Thus a cluster is like a structure, but it is a set of variables and not a piece of memory.

More specifically, the members of the cluster can be named by adding either a member name preceded by '`.`' or a '`[]`' bracketed list of integer constant subscripts to either the base name of the cluster or to another member name of the cluster. Two variable names with different base names belong to different clusters. Base names must refer to independent blocks of memory (2.4^{p15}), and are different if and only if the memory blocks they refer are not at the same memory location (and therefore being independent do not overlap).

Clusters can be passed to functions and returned from functions. For example:

```
define type pointer pair = subtype of void
    // if pointer pair pp then
    //   pp.begin points at the first element
    //   pp.end points just AFTER the last element

// Declaration of out of line function to allocate
// a vector of n T's.
//
( pointer pair pp -> T,
  unchecked adr pp.begin -> T,
  unchecked adr pp.end -> T )
    cluster allocate ( type T, basic(T) descriptor, uns32 n )

// Prefix operator to dereference a pointer pair.
//
( adr -> T V ) inline "*"
```

```
            ( type T,
              pointer pair pp -> T,
              unchecked adr pp.begin -> T,
              unchecked adr pp.end   -> T )
        unchecked:

        if ( pp.begin < pp.end ) return * pp.begin
        else fatal error
            ( "Deferencing empty pointer pair." )

    // Prefix operator to increment the begin pointer of a pointer
    // pair.
    //
    ( pointer pair pp -> T ) inline "++"
            ( type T,
              basic(T) descriptor,
              pointer pair pp -> T,
              adr -> unchecked adr pp.begin -> T )
        unchecked:

        unsadr c = pp.begin;
        pp.begin = c + descriptor.length;

    //
    // Example usage:
    //
    void my function ( void ):

        // Vector of 2 int32's is allocated and the elements
        // are set equal to 100 and 101.
        //
        pointer pair pp = allocate ( 2 ) -> int32
        * pp = 100
        * ++ pp = 101
        . . . .
```

Several heretofore unmentioned language features appear in this code.

First, cluster member names are used as prototype parameter names, with the base of these names also being a prototype parameter. This specifies that some parameters are related by being in the same cluster.

When this is done for arguments, the cluster base must also be an argument, and the cluster member values will be derived from the base, and must be omitted in calls.

When this is done for results, the cluster base may be either an argument or a result, and the values of the designated members of the cluster are set from the results. If the function has the '**cluster**' qualifier, then any returned cluster members that do not already exists are created as variables in the local code block, as is done by '`allocate`' above.

Also, a result parameter may have the same name as an argument parameter, as is done for the `pp` parameter in the prototype of `"++"` above. This specifies that the result parameter is the same as the argument parameter, and is merely being moved from argument to result in order to make the syntax work.

The above example could use a different implementation of `"++"` as follows:

```
// Unchecked prefix operator to increment an adr -> T value.
//
( adr -> adr -> T ) inline unchecked "++"
        ( type T,
          basic(T) descriptor,
          adr -> adr a -> T )
    unchecked:

    unsadr b = a;
    a = b + descriptor.length;
    return & a

// Prefix operator to increment the begin pointer of a pointer
// pair.
//
( pointer pair pp -> T ) inline "++"
        ( type T,
          pointer pair pp -> T,
          adr -> unchecked adr pp.begin -> T )
    unchecked:

    ++ pp.begin
```

Here an `unchecked "++"` prefix operator is first implemented after the manner of the C programming language for `adr -> T` values, and then this is used to implement the `"++"` operator for pointer pairs.

2.13 Arrays

An array is a cluster that can be used to access many variables whose addresses are computable by adding an integer to a pointer. The integer to be added is computed from other integers, usually by a linear function, though quadratic functions can also be used.

In theory arrays do not have to be built into the language, but they are because they are so widely used, and because loop code optimization is done specially for arrays, making it necessary to standardize the array interface.

They pointer type used as by an array is arbitrary, so long as it has an unchecked `"+"` operator that adds an integer to a pointer, and `.get.` and/or `.set.` operations to access values pointed at by a pointer.

The base of an array cluster is a pointer at the element type. If the base cannot be used directly as a pointer, it should have an '`unchecked`' access qualifier.

The members of an array cluster are:

.dimensions Type: any unsigned integer type. The number of dimensions (subscripts) of the array.

.step[i] Type: any integer type. The multiplier of the $i+1$'st subscript of the array for the purpose of determining the address offset of the element in the array. An integer (may be negative). The unit of these multipliers is the size of the array element.

.lower_bound[i] Type: any integer type. The lower bound (smallest legal value) of the $i+1$'st subscript of the array. An integer (may be negative).

.upper_bound[i] Type: any integer type. The upper bound (largest legal value) of the $i+1$'st subscript of the array. An integer (may be negative).

A number of operations that access array elements are normally defined. The following is an example:

```
// Define an array as a subtype of adr.
//
define type array = subtype of adr
```

```
// Allocate a 2D array.
//
( array a -> T,
  int32 a.dimension = 2,
  int32 a.upper_bound[0],
  int32 a.upper_bound[1],
  int32 a.lower_bound[0],
  int32 a.lower_bound[1],
  int32 a.step[0],
  int32 a.step[1] )
    new_array ( type T,
                int32 upper_bound_0,
                int32 upper_bound_1,
                int32 lower_bound_0 = 1,
                int32 lower_bound_1 = 1,
                int32 step_0 = upper_bound_1
                             - lower_bound_1 + 1,
                int32 step_1 = 1 )

// Return an element of a 2D array.
//
( adr -> T V ) ( type T, array a -> T,
                 a.dimension == 2,
                 int32 a.upper_bound[0],
                 int32 a.upper_bound[1],
                 int32 a.lower_bound[0],
                 int32 a.lower_bound[1],
                 int32 a.step[0],
                 int32 a.step[1] )
               [ int32 i0, int32 i1 ]
    unchecked:

    assert ( a.lower_bound[0] <= i0
             &&
             i0 <= a.upper_bound[0] )
    assert ( a.lower_bound[1] <= i1
             &&
```

```
            i1 <= a.upper_bound[1] )
    adr base = a -> T
    return base + a.step[0] * i0 + a.step[1] * i1

// Sample usage

   . . .
   array m -> float64 = new_array ( 100, 300 )
   m[3,5] = 3.141529
   float64 y = m[3,5]
```

TBD

type-name variable-name `[` *dimension-spec* { `,` *dimension-spec* }* `]`

where

dimension-spec ::= *bound-spec* *step-spec-option*

bound-spec ::= *length* | *lower-bound* `..` *upper-bound*

step-spec ::= `++` *step*

For example,

```
int32 iarray[5]
int32 adr piarray[1 .. 5] -> int32
adr -> int32 irarray[10,20]
adr -> int32 tirarray[10++1,20++10]
```

Here

- `iarray[i]` is an `int32` variable for `i` = `0`, `1`, `2`, `3`, and `4`. The `length`, which is `5`, specifies the number of elements in the array's one dimension, and array indexing begins at `0`.
- `piarray[i]` is a 'direct address of `int32`' variable for `i` = `1`, `2`, `3`, `4`, and `5`. The *lower-bound* and *upper-bound* of the dimension index, `1` and `5` respectively, are given.
- `& irarray[i,j]` is a 'direct address of `int32` variable', and `irarray[i,j]` is the `int32` location so addressed, for `i` = `0`, `1`, ..., `9` and for `j` = `0`, `1`, ..., `19`, where the two dimension *lengths* are given, respectively `10` and `20`. The elements of the array are stored in `10*20` consecutive memory locations that in memory order have indices

`(0,0), (0,1), ..., (0,19), (1,0), (1,1), ..., (9,19)`, with the last index `j` varying most rapidly. This means that the implied step size of the first dimension is `20` and of the second dimension is `1`.

- `tirarray[i,j]` is just like `irarray[i,j]` but the elements are stored in memory in transposed order: `(0,0), (1,0), ..., (9,0), (0,1), (1,1), ..., (9,19)`, with the first index `i` varying most rapidly. The step size of the first dimension is given as `1` and of the second is given as `10`.

An array is a cluster such that

$$\begin{aligned}\text{address of } array\texttt{[}i_0\texttt{,}i_1\texttt{,...,}i_{n-1}\texttt{]} = \\ & array\texttt{.base} \\ + \quad & array\texttt{.step[0]} * i_0 * \textit{element-size} \\ + \quad & array\texttt{.step[1]} * i_1 * \textit{element-size} \\ + \quad & \ldots\ldots \\ + \quad & array\texttt{.step[}n-1\texttt{]} * i_{n-1} * \textit{element-size}\end{aligned}$$

where

$$\begin{aligned}&array\texttt{.lower\_bound[0]} \le i_0 \le array\texttt{.upper\_bound[0]} \\ &array\texttt{.lower\_bound[1]} \le i_1 \le array\texttt{.upper\_bound[1]} \\ &\ldots\ldots \\ &array\texttt{.lower\_bound[}n-1\texttt{]} \le i_{n-1} \le array\texttt{.upper\_bound[}n-1\texttt{]}\end{aligned}$$

The members of an array cluster are called the ***parmeters*** of the array. The parameters of an array A whose element type is T and whose inheritable qualifiers are Q are

.first Type: '$Q\ T$ `*&`'. The first element of the array, that is, the lowest addressed array element that can be accessed using subscripts within the range indicated by the subscript bounds. '`&` A`.first`' is `NULL` if there are no accessible elements (i.e., if some upper bound is not at least as great as the corresponding lower bound).

.length Type: '`unsadr`'. The total number of elements in the array. The last used element of the array is

`(` $Q\ T$ `*&)` A`.first @ (` A`.length - 1 ) *` *element-size*

This is an unchecked expression that could be written to access the last array element. The `.length` is `0` if there are no array elements.

.base Type: '`noaccess` T `* (unchecked || readonly) * &`'. The address of the `[0,0,...,0]` element of the array. This may not actually be inside the array, as `0` subscripts may not be within bounds.

It is possible for unchecked code to store into the `.base` in order to relocate the array. When an array is passed to or returned from a function, it is actually the `.base` that is passed or returned.

.dimensions Type: any unsigned integer type. The number of dimensions (subscripts) of the array.

.step[i] Type: any integer type. The multiplier of the $i+1$'st subscript of the array for the purpose of determining the address offset of the element in the array. An integer (may be negative). The unit of these multipliers is the size of the array element.

.lower_bound[i] Type: any integer type. The lower bound (smallest legal value) of the $i+1$'st subscript of the array. An integer (may be negative).

.upper_bound[i] Type: any integer type. The upper bound (largest legal value) of the $i+1$'st subscript of the array. An integer (may be negative).

All the parameters except `.first` and `.length` are given to describe an array. The `.first` and `.length` parameters are then computed from the other parameters.[4]

Given the example from above,

```
int32 iarray[5]
int32 * piarray[1 .. 5]
int32 * & irarray[10,20]
int32 * & tirarray[10++1,20++10]
```

we have

```
iarray.lower_bound[0] == 0
iarray.upper_bound[0] == 4
iarray.step[0] == 1
iarray.length == 5

piarray.lower_bound[0] == 1
piarray.upper_bound[0] == 5
piarray.step[0] == 1
```

[4]Note that these are functions and not C language members, and as such only `.first` returns an lvalue in the sense of the C language.

```
piarray.length == 5

irarray.lower_bound[0] == 0
irarray.upper_bound[0] == 9
irarray.step[0] == 20
irarray.lower_bound[1] == 0
irarray.upper_bound[1] == 19
irarray.step[1] == 1
irarray.length == 200

tirarray.lower_bound[0] == 0
tirarray.upper_bound[0] == 9
tirarray.step[0] == 1
tirarray.lower_bound[1] == 0
tirarray.upper_bound[1] == 19
tirarray.step[1] == 10
tirarray.length == 200
```

When an array is allocated to the stack or to static memory, the length of the array is just enough to include all the elements of the array. There can be unused elements. Thus given

```
int32 x[1 .. 4 ++ 3]
```

then in memory the array `x` is laid out as

`x[1]`, unused, unused, `x[2]`, unused, unused, `x[3]`, unused, unused, `x[4]`

and we have `x.length == 10`, `x.first == x[1]`, `x.base == &x[1] - 3`.

The parameters of an array can be any expression that evaluates to an integer at the time the array is allocated. For example,

```
uns32 constant number = ...
int64 constant first = ...
uns32 constant step = ...

int32 x[number]
int32 y[first .. first + number - 1 ++ step]
```

The variables used to determine the parameters of an array must be '`constant`' or '`readonly`' so that checked code cannot modify them. More on this below.

2.13.1 Array Function Parameters

An array can be passed to a function and returned as the value of a function. Some examples are:

```
void sort ( int64 vector[length], int32 length )
void transpose ( float64 array[lb .. ub ++ step1, lb .. ub ++ step2],
                 int64 lb, int64 ub, int64 step1, int64 step2 )
(float64 array[length,length]) unit ( uns32 length )
(string output[olength], uns32 olength) sort
        ( string input[ilength], uns32 ilength )
```

When an array is passed to a function, the parameters needed to describe the array must be expressions computable from parameters passed to the function. When an array is returned by a function, the parameters needed to describe the array must be expressions computable from parameters either passed to or returned from the function. Note that integer function parameters are '`constant`' by default.

When a function that takes arrays as input parameters or computes arrays as output results is called, the array parameters that are function parameters or results may be implicitly specified. Examples using the function declarations above are:

```
int64 x[1 .. 100]
. . . compute x . . .
sort ( x[] )    // length implicitly passed.
float y[1 .. 5][1 .. 5]
. . . compute y . . .
transpose ( y[] )     // lb, ub, step1, step 2 implicitly passed.
float z[] = unit ( 10 )     // computes z and its parameters.
string s[10]
. . . compute s . . .
string t[] = sort ( s[] )     // ilength and olength implicitly passed.
                              // computes t and its parameters.
```

In order to indicate that implicit array parameter passing is desired, the empty brackets '[]' must be appended to the array name.

Checked functions can only allocate arrays to static or stack memory. Unchecked functions can be written to allocate arrays in a heap and return them.

The parameters of an array need not be constants: they can be variables. However, the variables must be qualified (see 2.14^{p39}) in such a way that checked code cannot modify

them. Unchecked functions may then be written to change the array parameter variables and reconfigure the array: e.g., the array may be expanded. An example is:

```
// Implement my array:

define type my array = (32,8)
readonly unsadr *& .n ( my array *& @ a ) unchecked = a@0
readonly unsadr *& .max_n ( my array *& @ a ) unchecked = a@8
readonly unsadr *& .inc_max ( my array *& @ a ) unchecked = a@16
noaccess float64 (readonly *&b) [1 .. a.n] .void ( my array *& @ a )
    unchecked = a@24
(my array *& @ a) .constructor.() unchecked
{
    a.n = 0
    a.max_n = 0
    a.inc_max = 30
    a.void.base = NULL
}
void push ( my array *& a, float64 value ) unchecked
{
    if ( a.n >= a.max_n )
    {
        a.max_n += a.inc_max
        float64 * ap =
            malloc ( sizeof ( float64 ) * ( a.max_n ) )
        if ( a.void != NULL )
            memcpy ( ap, a.void.base,
                     a.n * sizeof ( float64 ) )
        mfree ( a.void.base )
        a.void.base = ap
    }
    ap[a.n] = value
    ++ a.n

}

// Use my array:
```

```
my array a
push ( a, 1.0 )
push ( a, 2.0 )
a[1] // Equals 1.0.
a[2] // Equals 2.0.
a[0] // Run time error.
a[3] // Run time error.
```

There are a number of language features introduced in the above code.

First, `a@24` is of type '`void *&`' which is converted to type '`float64 (*&) [1 .. a.n]`' by the '`unchecked`' code of the `.void` function.

Second, an array value is just its `.base`, which is just a pointer. Viewed as the element of an object, `a.void` is just a `.base` pointer, and just enough space to store this pointer must be allocated to it. Here we treat pointers as being 8 byte long values that are aligned on 8 byte boundaries, but as 4 byte pointers will fit into 8 bytes, the code will work even with 4 byte pointers.

Third, unchecked code can store a pointer of type 'T `*`' into X`.base`. The line '`a.void.base = ap`' above does just this. Unchecked code can also read the base, as in the line '`memcpy ( ap, a.void.base, n )`', and implicitly convert the base to a pointer to a '`readwrite`' value. Checked code can also read the base, but the value returned will be given the type 'noaccess T `*`' since base pointers may not point at real memory (e.g., given X`[1 .. 2]` then X`.base` points to an unusable location just before the first element).

Fourth, '`unchecked`' code can use a pointer as if it were a 1-dimensional array with step `+1` and no bounds. This is done in the statement '`ap[a.n] = value`' above.

Fifth, '`.base`' when applied to an array, and '`.void`' applied to any expression, can always be omitted, as long as the result can be disambiguated. Thus in the above '`a.void.base`' could be replaced by either '`a.void`' or just '`a`'. In the code at the end after the functions, '`a.void[...]`' is replaced by '`a[...]`'.

Sixth, '`float64 (*&b) [1 .. a.n]`' differs from '`float64 *&b [1 .. a.n]`' which is equivalent to '`float64 *& (b [1 .. a.n])`'. The first subexpression means that the value returned is a pointer to the location holding the base of an array whose elements are `float64` values. The second subexpression means that the value returned is the base of an array whose elements are pointers to '`float64`' values (and `b[i]` refers to the '`float64`' value pointed at by the array element indexed by `i`). In the expression '`float64 (readonly *&b) [1 .. a.n]`' it is the array `.base` that is read-only, not the array elements.

Seventh, the code above contains both an implementation of `my array` and a use of `my array`. We remind the reader that the implementation is intended to be written by macros which are called by the end user and which ensure that the unchecked code generated is in fact type-safe, while the code that uses `my array` is intended to be written by the end user and should be type-safe in its own right.

2.13.2 Copying Arrays

If you want to copy entire arrays you have to apply the '`*`' operator to them. Thus

```
declare alias qualifier same_data
same_data int32 x[1 .. 10]
same_data int32 y[1 .. 10] = x
// Now x and y are the same piece of memory.  This would not be
// legal without the alias qualifier (described later).
y[1] = ..., etc.
int32 z[1 .. 10]
* z = * y
// Now the elements of x (and y) have been copied to
// the elements of z.
```

2.13.3 Subarrays

Subarrays can be computed by expressions of the form

array-expression `[` *selector-spec* { `,` *selector-spec* }* `]`

where

selector-spec ::= *lower-bound* `..` *upper-bound* *step-spec-option*

Here the bounds are valid subscripts for the given array, and the steps are increments in these valid subscripts.

For example,

```
declare alias qualifier first_half, second_half
first_half second_half int32 x[1 .. 10]
first_half int32 x1[1 .. 5] = x[1 .. 5]
second_half int32 x2[1 .. 5] = x[6 .. 10]
```

```
declare alias qualifier odd_half, even_half
odd_half even_half int32 y[1 .. 10]
odd_half int32 y1[1 .. 5] = y[1 .. 9 ++ 2]
even_half int32 y2[1 .. 5] = x[2 .. 10 ++ 2]
```

Here `x1` is the first half of `x` and `x2` is the second half. Similarly `y1` is all the elements of `y` with odd subscripts, and `y2` is all the elements with even subscripts.

The '`=`' operator used with arrays overlays and does not copy. It can only be used to initialize subarray names, and only requires that the arrays involved have equal numbers of subscripts in each dimension.

In somewhat more generality a subarray of an array may be given a ***linear view*** defined by any linear map between arbitrary subscripts and the subscripts of the array. For example:

```
declare alias same_data
same_data int32 x[1 .. 70]
same_data int32 y[1 .. 10, 1 .. 7] = ([i,j] ===> x[(i-1)*7 + j])
same_data int32 z[1 .. 7, 1 .. 10] = ([i,j] ===> y[j,i])
same_data int32 w[1 .. 6] = ([i] ===> y[i,i+1])
```

Here `y` is a 2-dimensional array overlaid on the vector `x`, `z` is the transpose of `y`, and `w` is a subdiagonal of `y`.

2.13.4 Array Maps

Special forms of array can be defined by adding parameters to the array and defining special operators. For example, a symmetric array can be defined by:

```
int32 x[28]
uns32 x.size = 7
int32 *& x[int32 i, int32 j]
{
    assert ( 0 <= i < size && 0 <= j < size )
    return x[i*(i+1)/2 + j]
}
```

Here '`.size`' is an extra member of the cluster `x` and '`x[ ]`' is just a function that surrounds it arguments with '`[ ]`' instead of '`( )`'.

2.14 Qualifiers

The L-Language controls the way memory is accessed via ***qualifiers***. A qualifier is either a *name* (p19), or is a parentheses surrounded qualifier expression.

Qualifiers may be applied to values or to locations of memory. Some types of qualifiers apply to values and some to locations. For example, '`stack`' is a value qualifier that says that a value is not valid after the currently executing function returns, '`constant`' is a location qualifier that says that the location will never ever be written into, and '`readonly`' is a location qualifier that says that the location cannot be written using the given address of the location.

For example, the declarations

```
constant int32 x = 5
stack adr yp = & x -> readonly int32 y
```

promise that after initialization no one will ever write into the location `x` (which is `constant`), that `yp` will be valid as long as the current function execution exists (because it holds a `stack` value), and that `yp` cannot be used to write to `y` (which is `readonly`).

A location qualifier is applied to the type of a value that is the target of a pointer. That is, they appear in types of the form

pointer-type `->` *location-qualifier value-type*

Location qualifiers actually affect operations on the pointer, and not on the value pointed at. Thus both '`constant`' and '`readonly`' applied to the pointer target type prevent the builtin `.set.` operator from being executed on the pointer itself.

Qualifiers may be given in type definitions, and apply to all values and memory locations of the defined type. For example, if `Q` is a qualifier,

```
define type Q type1 = (48,8)
type1 x
adr yp -> type1
```

is equivalent to

```
define type type1 = (48,8)
Q type1 x
adr yp -> Q type1
```

Qualifiers may also be applied to code blocks. Such are called '***code block qualifiers***'. A qualifier just before the `=` or `{` that introduces the function body in a function definition

applies to the code in the body. For example, if `Q` is a qualifier,

```
int32 function1 () Q = ...
int32 function2 () Q
{
   . . .
}
```

apply `Q` to the body of the functions.

A qualifier just before the name of a function in a function definition or declaration applies to calls to the function, and is said to be a '***function access qualifier***'. For example,

```
int32 Q function3 ()
( adr -> int32 V ) Q function4 ()
```

Some qualifiers are both access and body qualifiers, and when these are put in the position of function access qualifier, they also apply to the function body.

If in the above examples we assume that `Q` is a protection qualifier, then `function3` and `function4` can only be called by function bodies (or code blocks, see below) that have the `Q` qualifier, and in particular these functions can be called by `function1` and `function2`.

A qualifier immediately preceding a `{ }` bracketed code block inside a function applies to the code block. For example,

```
int32 function5 ()
{
   . . .
   Q {
       . . . a code block . . .
   }
}
```

applies `Q` to the code block. If `Q` is an access qualifier, the code block can call functions with access qualifier `Q`, even though the body of `function5` outside the code block cannot.

There are several kinds of qualifiers. We summarize these here, and give details in following sections.

Lifetime qualifiers specify the lifetime of a value. This in turn controls where the value may be stored, e.g., a value of limited lifetime may not be stored in a location whose value is declared to be of unlimited lifetime.

There are three builtin lifetime qualifiers. The '`static`' qualifier specifies that the value has

unlimited lifetime. The '`stack`' qualifier specifies that the value has a lifetime that lasts at least until a current code block execution terminates. The '`caller`' qualifier specifies that the value has a lifetime that begins before the called function starts to execute and lasts until sometime after that function returns.

In addition user define 'calling lifetime qualifiers' may be defined that specify that lifetimes will continue until an out-of-line function having a given access qualifier (see below) is called. For example, a '`relocatable`' lifetime qualifier can specify that a value lifetime ends when a function with the '`relocating`' access qualifier is called.

Caching qualifiers specify when memory locations may be changed, and thence when they may be cached in registers. The '`constant`' qualifier specifies that a location will never be changed, and may be cached indefinitely in registers. The '`unique`' qualifier specifies that a location has no overlaps with other locations, and, in particular, that a location passed from a caller to a called function does not overlap with any other location so passed or with any location accessible from other data.

In addition user define 'calling cache qualifiers' may be defined that specify that software cache lifetimes will continue until a code block having a given access qualifier (see below) is executed. If such a code block flushes hardware caches, it can be used in combination with these user defined qualifiers to flush both software and hardware caches.

Cachine qualifiers, types and container types, and regions are used together to specify aliasing ($??^{p}??$).

Access qualifiers may be declared to control which functions can access a memory location or call other functions. Access qualifiers may or may not be ***protection qualifiers***. The builtin non-protection access qualifiers are '`readwrite`', '`readonly`', '`writeonly`', and '`noaccess`'. The builtin protection qualifier is '`unchecked`'. Protection qualifiers may also be defined by the user.

A memory location with protection qualifier X can only be accessed by code blocks that have qualifier X, and a function with an access protection qualifier X can only be called by functions whose bodies have qualifier X. The builtin protection qualifier '`unchecked`' permits access to various builtin operations that explicitly or implicitly involve type conversion.

Memory locations can be assigned qualifier expressions that are logical combinations of qualifier names and the operators '`|`' and '`&`'. For example, a memory location may be assigned the expression '`(`X`|constant)`' where X is a protection qualifier. A function body with qualifier X will be able to access such a memory location to both read and write it, but a function body without qualifier X will treat the location as having the '`constant`' qualifier, will not be able to write the location, and will assume that the location will not

be written by others. In order for this to work properly, the functions with qualifier X that are permitted to write the location must be called first, before functions without qualifier X are allowed to access the location. Alternatively, '`constant`' can be replaced by '`readonly`', which forbids writing the location but does not promise that other code will not write the location.

Inline qualifiers specify that functions called from a particular block of code are to be inlined, or that a particular block of code is to be out-of-line even if code surrounding it is being inlined. Inlining is necessary to get full optimization.

Qualifiers may also be defined that represent ***groups*** of qualifiers.

Qualifiers may also be ***inherited***. For example, if `G` is a group qualifier,

```
G? readonly unsadr .n ( adr -> my array G? a ) unchecked = ...
```

causes any qualifier in group `G` that is present for the argument '`a`' to be applied to the function result. See 2.14.7^{p55}.

2.14.1 Lifetime Qualifiers

Lifetime qualifiers determine when value lifetimes expire. For example, the lifetime of the address of a stack variable allocated in a code block frame expires when the code block execution terminates.

The '`static`' lifetime qualifier that indicates a value has an unlimited lifetime is applied by default to numeric values. But it is possible to apply other lifetime qualifiers. For example, if an `int32` value is used as an open file designator, it may be given a limited lifetime.

A lifetime qualifier `Q1` is said to be '***convertible***' to a lifetime qualifier `Q2` if `Q1` promises a lifetime at least as long as `Q2` promises. Thus the '`static`' lifetime qualifier is convertible to any other lifetime qualifier.

If `Q` is a lifetime qualifier in the type

```
adr xp -> Q mytype x
```

then `Q` is said to be a qualifier of the location `x`, all values written into that location must have lifetime qualifiers that are convertable to `Q`, and all values read from that location will have qualifier `Q`.

Three lifetime qualifiers are builtin: '`static`', '`stack`', and '`caller`'. Checked code can only allocate to static memory or to the stack memory of the currently executing function, and

addresses of variables allocated by checked code are given the '`static`' or '`stack`' qualifier accordingly. Addresses of variables that are part of a function argument or return value are given the '`caller`' lifetime qualifier by default.

static Specifies the value lives forever. Automatically applied to the addresses of locations allocated at load time, and may be applied to the addresses of locations allocated at program initialization time and never deallocated.

The '`static`' lifetime qualifier is convertible to any other lifetime qualifier. It is also the default lifetime qualifier for non-pointer values.

stack Specifies the value lives as least as as long as the current function execution. Equivalent to `stack(0)`: see the following.

'`static`' values are convertable to '`stack`' values.

stack(n) Specifies the value lives at least as long as the execution of the current code block of nested level `n` within the currently executing function. Unnested code within the function body is considered to be at nested level `0`. `stack(n)` is automatically applied to the addresses of locations allocated to the current frame by code in a level `n` code block (such locations are freed when the code block of level `n` terminates).

'`stack(m)`' is convertable during the execution of a code block of level `n`>`m` to '`stack(n)`'.

A '`stack(n)`' qualifier cannot be applied to function argument and return types.

caller Only used in a function F's argument or return value types to qualify a values such as addresses passed to the function. Specifies the value's lifetime begins sometime before a call to F and continues until sometime after that call returns.

The '`caller`' qualifier is like the `stack` qualifier but specifies that the value is valid at the moment the function begins executing and at the moment the caller code returned to by the function begins executing.

'`static`', '`stack(n)`', and `caller` qualifiers can be converted to the '`caller`' qualifier when the latter is part of the type of a value that is being passed as an argument or a return value to a function being called by the code that does the conversion. However, '`stack(n)`' and `caller` qualifiers cannot be converted to the '`caller`' qualifier when the latter is part of the type of a value that is an argument or return value of the currently executing function. Note that in this case a '`caller`' qualifier cannot even be converted to a '`caller`' qualifier.

For example:

```
int32 function1 ( caller adr arg1 -> static adr -> int32 )
```

```
    // The argument is a pointer to a caller allocated location
    // holding a pointer to a statically allocated int32.
    // function1 can use ‘*arg1 = value’ to return a static
    // adddress to an int32 value, and can also use
    // ‘**arg1 = value’ to return an int32 value.
int32 function2 ( caller adr arg1 -> caller adr -> int32 )
    // The argument is a pointer to caller allocated location
    // holding a pointer to a caller allocated location holding
    // an int32.  function2 can use ‘**arg1 = value’ to return
    // an int32 value, and can use ‘*arg1 = value’ to return
    // a static address, but not a caller address.
int32 function3 ( static adr arg1 -> stack adr -> int32 )
    // Illegal; an argument type cannot contain a ‘stack’
    // lifetime qualifier.
```

The ‘`caller`’ lifetime qualifier is the default lifetime qualifier for any function argument or return value pointer value that has no lifetime qualifier. Thus in the above example the ‘`caller`’ qualifiers could have been omitted.

The ‘`caller`’ lifetime qualifier may not be used except in function argument and return value types.

Users may define ***calling lifetime qualifiers*** by using a declaration of the form:

```
declare calling lifetime qualifier qualifier-name
        with continuer qualifier-name
        with discontinuer qualifier-name
        with default qualifier-name
```

An example is:

```
declare calling lifetime qualifier relocatable
        with continuer nonrelocating
        with discontinuer relocating
        with default relocating
```

If `Q` is a calling lifetime qualifier, `CQ` is its continuer, and `DQ` is its discontinuer, then a value with qualifier `Q` will have a lifetime that is terminated by a call to an out-of-line function that has the access qualifier `DQ` or the execution of a code block that has qualifier `DQ`.

If an out-of-line function has the `CQ` access qualifier, the lifetime of a value with qualifier `Q` is not terminated by a call to the function. Similarly if a code block has the `CQ` qualifier, the lifetime of the value is not terminated by the execution of the code block.

The default qualifier is applied to out-of-line functions with no explicit `CQ` or `DQ` access qualifier. The `CQ` qualifier is applied by default to both code blocks and to inline functions with no explicit `CQ` or `DQ` qualifier.

The `DQ` and `CQ` qualifiers are incompatible with each other.

The discontinuer lifetime qualifier `DQ` is also automatically declared to be a protection qualifier(p51) so that a block of code with continuer qualifier `CQ` cannot call a function or invoke a block of code with the discontinuer qualifier `DQ`.

Thus in the example, a '`relocatable`' value would have a lifetime until an out-of-line function with no '`nonrelocating`' access qualifier is called, or a code block with the '`relocating`' qualifier is executed, where this code block could be part of an inline function.

2.14.2 Caching Qualifiers

When a memory location is moved into a register, the register becomes a ***software cache*** of the memory location. Thus memory locations are cached in registers under the control of software, and not hardware. Unfortunately this makes things very difficult for the compiler (the ultimate fix is a radical change in computer architecture; see Appendix A^{p58}).

Memory locations may also be placed in hardware caches. In a hardware system with memory shared among multiple central processing units, each central processing unit may have its own hardware cache that is not automatically synchronized with shared memory or the caches of other central processing units. Special instructions are used to synchronize these hardware caches with shared memory.

Caching qualifiers determine when to invalidate and when to write software and hardware caches.

Two caching qualifiers are builtin: '`constant`' and '`unique`'. Two kinds are declarable: alias qualifiers and parallel qualifiers.

constant The memory location is constant and will not be modified at all.

The memory location can be cached in registers without restriction.

Memory locations declared '`constant`' must be written during their initialization. This is managed by not declaring a location to be '`constant`' when it is passed as a result value to the function that initializes the location.

The following code has some examples:

```
int32 function1 ( stack adr -> constant int32 arg1 )
int32 function2 ( stack adr -> readonly int32 arg1 )
int32 function3 ( stack adr -> int32 arg1 )
. . . . . .
constant int32 value1 = 5
   // Permitted:
   //   caller adr -> int32 .assign. ( 5 ) is called.
... my_function ( ... )
{
    value1 = 10
       // Erroneous:
       //   caller adr -> constant int32 .assign. ( 10 ) is called.
    function1 ( value1 )   // Permitted; arg1 constant.
    function2 ( value1 )   // Permitted; arg1 readonly.
    function3 ( value1 )   // Erroneous; arg1 NOT constant
                           // or readonly.
    int32 value2 = 15
    function1 ( value2 )   // Erroneous.  Function1 might make
                           // a long term cache of value2 which
                           // might then be changed.
    function2 ( value2 )   // Permitted.  Function2 merely
                           // promises not to write value2.
    . . . . . .
```

Note that the '`constant`' qualifier in an argument type is a promise that no code will ever modify the location. By contrast a '`readonly`' argument qualifier (like '`const`' in C/C++) is a promise only that the location will not be modified during the course of the call. The difference is that a called function may construct a long-lived cache of a '`constant`' location but not of a '`readonly`' location.

The '`constant`' qualifier obeys the following special rules:

1. '`constant`' is removed from a location passed to a function as the result location in the initializer statement of the location.
2. '`constant`' is the default cache qualifier for locations of function arguments that are passed by value and not by reference.

The '`constant`' qualifier is automatically applied to temporary values. Thus continuing

the above example:

```
function1 ( value1 + value2 )
    // Permitted; value1 + value2 is a 'constant' int32
    // temporary value.
```

unique Function argument and return value locations can be given the `unique` qualifier if these locations are being passed to the function from its caller. These locations cannot overlap any other location passed to the function or available to the function via global data. This means these locations must either have been allocated by the caller, or passed as '`unique`' argument or return value locations to the caller from its caller.

A '`unique`' memory location may be passed as an argument to or return value of a function only if the function argument or return value location is declared '`unique`' within the function declaration. A stack memory location may be so passed if no address that permits access to the location has been stored anywhere except in the stack and the location is not passed as more than one of the function argument or return value locations.

Addresses that permit a '`unique`' memory location to be accessed may only be stored in stack locations where they can be tracked by the compiler.

A '`unique`' memory location may be cached in a register. This cache must be flushed when an out-of-line function is called only if the address of the location is passed to the out-of-line function as the address of a '`unique`' function argument or return value location.

Alias Qualifiers Alias qualifiers are declared by the programmer and are used to tag memory locations that may be aliased with each other.

At compile time the L-Language converts location addresses into a form that is the sum of a ***base address*** and an integer ***offset***. In addition the location may have associated alias qualifiers. The L-Language assumes two locations do not overlap unless they have a common alias qualifier, or unless the locations have identical base addresses and their offsets indicate they overlap.

A base address is one of the following:

1. The address of a memory block allocated by the loader.
2. The address of a memory block allocated to the current function frame.
3. An address passed into the current function as the location of an argument or return value.

4. An address returned from a function call or builtin operation, including addresses read from memory locations.

For example:

```
void copy1 ( uns8 source[length],
             uns8 destination[length],
             uns32 length )
{
    uns32 i = 0
    loop: {
        if ( i >= length ) break loop
        destination[i] = source[i]
        ++ i
    }
}
declare alias qualifier same_data
void copy2 ( same_data uns8 source[length],
             same_data uns8 destination[length],
             uns32 length )
{
    uns32 i = 0
    loop: {
        if ( i >= length ) break loop
        destination[i] = source[i]
        ++ i
    }
}
uns8 x[16]
x[0] = 0
copy1 ( x[0 .. 14], x[1 .. 15], 15 )   // Erroneous.
copy2 ( x[0 .. 14], x[1 .. 15], 15 )   // Zeros x.
```

The call to `copy1` is erroneous because arguments that are not supposed to be aliased are in fact aliased.

Aliasing is checked as necessary by the compiler and at run time. The run-time checks can be suppressed by compilation switches.

Simple alias qualifiers are just names. Complex alias qualifiers contain a ***type overlap***

expression as follows:

complex-alias-qualifier ::= *alias-name* `(` *type-overlap-expression* `)`

type-overlap-expression ::= *type-inclusion-sequence* { `|` *type-inclusion-sequence* }*

type-inclusion-sequence ::= *type-factor* { `>>` *type-factor* }*

type-factor ::= *type* | `(` *type-overlap-expression* `)`

A canonical *type-overlap-expression* is one with no parenthesized *type-overlap-subexpressions*, and any *type-overlap-expression* can be made canonical by distributing `|` over `>>` as in

$$\ldots \texttt{>>} (\ e_1 \texttt{|}\ e_2\ ) \texttt{>>} \ldots \quad \Longrightarrow \quad \ldots \texttt{>>}\ e_1 \texttt{>>} \ldots \texttt{|} \ldots \texttt{>>}\ e_2 \texttt{>>} \ldots$$

A *type-inclusion-sequence*

$$t_1\texttt{>>}t_2\texttt{>>}t_3\texttt{>>}\ldots\texttt{>>}t_n$$

intuitively describes a hierarchical data block structure: a date block b_1 of type t_1 contains a subblock b_2 of type t_2 and this contains a subblock b_3 of type t_3, etc., until we get to the smallest data block b_n of type t_n which in turn contains the location being qualified. Furthermore, b_1 is allocated to the stack memory, to static memory, or to a heap memory, and is disjoint from every other data block so allocated that has a type incompatible with t_1. Similarly b_2 is a child of b_1 in the hierarchical containment scheme, and is disjoint from every other child subblock of b_1 that has a type incompatible with t_2. Etc. through b_n and t_n.

Formally two data types overlap if they are compatible, which means that a location of one type may be viewed as having the other type. Two *type-inclusion-sequences* overlap if one is an initial segment of the other, where compatible types are viewed as being equal. For example '`mydata>>subdata1>>int`' and '`mydata>>subdata1`' overlap but '`mydata>>subdata1>>int`' and '`mydata>>subdata2>>int`' do not, assuming that `subdata1` and `subdata2` are incompatible types.

Note that '`type1>>type2`' refers to a datum of `type2` located inside a datum of `type1` without any intervening data; i.e., there is no `type12` such that '`type1>>type12>>type2`' can also be used to describe the datum.

Intuitively a canonical *type-overlap-expression* $e_1 | e_2$ describes a data block that can be described by either of the *type-inclusion-sequences* e_1 or e_2. Formally, two canonical *type-overlap-expressions* overlap if any *type-including-sequence* from the first overlaps any *type-including-sequence* from the second.

TBD: include fixed integer offsets as part of inclusion?

Parallel Qualifiers Parallel qualifiers are declared by the programmer and are used to tag memory locations that may be flushed from software and hardware caches in order to permit separate central processing units to communicate through shared memory.

There are three L-Language statements that flush caches. Each gives a list of parallel qualifiers that names all the locations accessible at the point of the statement that are tagged with these qualifiers. An example is:

```
declare parallel qualifier pdata, pflag
pdata float64 x[3]
pflag bool done_flag = false
. . . . . .
// Output data
x[0] = 35.87
x[1] = -2.90
parallel flush write pdata
// Set done_flag indicating data has been output
done_flag = true
parallel flush write pflag
// Wait for data to be consumed
// Consumer clears done_flag after consuming data
loop: {
    spin ( 5000 ) // 5000 nanosecond spin
    parallel flush read pflag
    if ( done_flag == false ) break loop
}
// Data has been consumed
```

The '`parallel flush`' command flushes both software and hardware caches.

2.14.3 Access Qualifiers

Five access qualifiers are builtin: '`readwrite`', '`readonly`', '`writeonly`', '`noaccess`', and '`unchecked`'. The last two are protection qualifiers; additional protection qualifiers may be declared.

readwrite, **readonly**, **writeonly**, **noaccess** Specifies whether execution may read or write a location.

unchecked A builtin protection qualifier that is required to access the builtin operators that perform explicit or implicit type conversions.

Protection Qualifiers Protection qualifiers may be declared by the programmer. Protection qualifiers may have names of the form

protection-qualifier-name `(` *type-expression* `)`

A code block may call a function only if every protection qualifier of the function is also protection qualifier of the code block, or the special '`unchecked`' qualifier is a qualifier of the code block. Similarly a code block may access a memory location only if every protection qualifier of the location is also a protection qualifier of the code block, or the special '`unchecked`' qualifier is a qualifier of the code block.

Often qualifier expressions containing protection qualifiers are used as function and memory location access qualifiers. For this reason, examples of access qualifiers are given in the next section.

2.14.4 Qualifier Expressions

A ***qualifier expression*** is an expression composed of qualifiers, the operators '`|`' and '`&`', and parentheses. Qualifier expressions can be used as function access qualifiers and memory location qualifiers.

In order for a function to be callable or memory location to be accessible by a code block, all of the function's access qualifier expressions or the location's qualifier expressions must evaluate to true according to the following rules. First, each protection qualifier is given the value true if it qualifies the code block and false otherwise. Second, every non-protection qualifier is given the value true if and only if doing so is necessary and sufficient to make all the subexpressions containing the non-protection qualifier true. This last rule is applied from left to right. Any non-protection qualifiers given the value true for any qualifier expression then qualify the function call or location when it is accessed by the code block, while non-protection qualifiers given the value false are ignored.

For example, given the code

```
define type type1 = (24,4)
declare protection qualifier constructor(type1)
(constructor(type1)|constant) type1 my_data
void function1 ( type1 *& arg1 )
void function2 ( constant type1 *& arg1 )
void function3 ( type1 *& arg1 ) constructor(type1)
```

```
. . . . . .
function1 ( my_data)    // Erroneous; my_data is constant for
                        // function1.
function2 ( my_data)    // Legal; my_data is constant for
                        // function2 and so is arg1.
function3 ( my_data)    // Legal; my_data is NOT constant for
                        // function3.
```

Note the left-to-right clause in the rule for evaluating non-protection qualifiers. For example, given the code:

```
define type type1 = (24,4)
declare protection qualifier A
((A&readonly)|constant) type1 my_data1
((A&readonly)|(A&readwrite)|constant) type1 my_data2
```

then to a function with protection qualifier `A`, `my_data1` is `readonly` and not `constant`, whereas to all other functions, `my_data1` is `constant` and not `readonly`. `My_data2` behaves just like `my_data1`, as the subexpression '`(A&readwrite)`' can never have any effect.

The precise specification for evaluating qualifier expressions is as follows:

1. A protection qualifier evaluates to true or false according to whether the invoking code block has or does not have the qualifier.

2. Any other qualifier evaluates to true when it is evaluated. When this happens the qualifier is asserted for the function access or location qualified by the qualifier expression. Note that evaluation may skip over subexpressions and thereby not evaluate qualifiers therein.

3. Evaluation of the subexpression $x|y$ evaluates x first. If x evaluates to true, the subexpression evaluates to true and evaluation of y is skipped. If x evaluates to false, y is evaluated and its value becomes the value of the subexpression. Note that `|` is left associative: $x|y|z \equiv (x|y)|z$.

4. Evaluation of the subexpression $x\&y$ evaluates x first. If x evaluates to false, the subexpression evaluates to false and evaluation of y is skipped. If x evaluates to true, y is evaluated and its value becomes the value of the subexpression. Note that `&` is left associative: $x\&y\&z \equiv (x\&y)\&z$.

2.14.5 Inline Qualifiers

To '***inline***' a function is to replace the code calling the function with the code of the function body. For optimal performance functions should be inlined, so that is the default.

Inlining is controlled by two builtin qualifiers: '**inline**' and '**outline**'. These are applied to function bodies and other code blocks. An '`inline`' code block is inlined and an '`outline`' block not inlined. An '`outline`' block may be nested inside an '`inline`' block, but an '`inline`' block may not be nested inside an '`outline`' block.

An example is:

```
void function1 ( int32 x ) inline { ... }
void function2 ( int32 x ) outline { ... }
void function3 ( int32 x ) inline:

    if ( x > 0 ):
        ... inlined code ...

    else outline:
        ... outlined code ...
```

Calls to `function1` are inlined, those to `function2` are not, while those to `function3` are inlined except for the part executed when `x<=0` which is treated as an out-of-line function.

Recursive inlining is permitted provided it does not lead to unbounded code. For example:

```
int32 function1 ( int32 n ) inline { ... }:
int32 recurse ( int32 n ) inline:
    if ( n == 0 ) return 0
    else return function1 ( n ) + recurse ( n - 1 )

int32 function2 ( void ) outline:
    return recurse ( 5 )
        // Works, inlining recurse 6 times.

int32 function3 ( int32 n ) outline:
    return recurse ( n )
        // Fails by producing an unbounded amount of code.
```

Inlining recursion works because while inlining code the compiler computes compile time constant expressions and uses the results to eliminate conditionalized code.

It is possible to control inlining to some degree by associating compile time constant integers with the '`inline`' and '`outline`' qualifiers. If a code block with qualifier '`outline(`m`)`' calls a function whose code block has qualifier '`inline(`n`)`', then the function code block is inlined if and only if $m \geq n$. Thus a higher m associated with and '`output`' code block inlines more code into the out-of-line block, and a higher n associated with an '`inline`' code block makes it less likely the code block will be inlined.

For example:

```
int32 function1 ( int32 n ) inline(1) { ... }
int32 function2 ( int32 n ) inline(2) { ... }
int32 function3 ( int32 n ) outline(1):

    function1(...) // Inlined.
    function2(...) // NOT inlined.

int32 function4 ( int32 n ) outline(2):

    function1(...) // Inlined.
    function2(...) // Inlined.
```

The numbers associated with the '`inline`' and '`outline`' qualifiers are called ***optimization indices***. Optimization indeces default to the value `1`; e.g., '`inline`' is equivalent to '`inline(1)`'.

2.14.6 Qualifier Transitivity and Defaults

A qualifier may be declared to be ***transitive qualifier***, meaning that if the qualifier is given as a function access qualifier it will automatically be attached as a code block qualifier of the function body.

A qualifier may be declared to be ***default function access qualifier*** meaning that it is applied as a function access qualifier by default to all functions that do not have a given ***counter qualifier***. The counter qualifier merely specifies that the default does not apply to a particular function.

Usually it is protection qualifiers that are declared to be transitive or default function access

qualifiers.

2.14.7 Qualifier Inheritance

2.15 Aliasing and Containers

For purposes of optimization it is important to be able to accurately determine if two pointers may point at overlapping data. If they do, data loaded into a register using one pointer may become invalid when the other pointer is used to write data. When two pointers point at overlapping data, they are said to '***aliased***'.

The L language uses the concept of type non-overlap and container types to track aliasing. Consider the pointer definitions:

```
define type Tx ...
define type TxC ...
define type Ty ...
define type TyC ...
. . . . .
adr xp -> Tx @ TxC x
adr yp -> Ty @ TyC y
```

If it is known that data of types `Tx` and `Ty` cannot overlap, then the pointers cannot be aliased. But this is also true if it is known that data of types `Tx` and `TyC` cannot overlap, or data of types `TxC` and `Ty` cannot overlap, or data of types `TxC` and `TyC` cannot overlap.

Type overlap is represented by two relations. The `T1 <@ T2` relation means that a datum of type `T1` can be completely contained inside a datum of type `T2`. This relation is assumed to be transitive and reflexive, and it is possible for both `T1 <@ T2` and `T2 <@ T1` without `T1` and `T2` being the same type. The second relation `T1 ~@ T2` means that a datum of type `T1` may overlap a datum of type `T2` without either data being contained in the other. This relation is assumed to be reflexive and symmetric but not transitive.

These relations are specified by including the following in the context of a code block that needs to determine whether two pointers are aliased. Whenever `T1 @ T2` appears in the context, `T1 <@ T2` is added to the `<@` relation. Whenever a type `T1` is defined to be a subtype of the type `T2`, then `T1 <@ T2` is added to the `<@` relation. The last of adding to the `<@` relation, and the only way of adding to the `~@` relation, is with statements of the form:

`declare overlap` *type-name* { `@` | `<@` | `<@>` | `~@` } *type-name*

where here `@` is taken to mean `<@` and `T1 <@> T2` is taken to mean that both `T1 <@ T2` and `T2 <@ T1`.

Then two types `T1` and `T2` are defined to be non-overlapping if none of the following is given data or deducible from given data using transitivity of `<@` or symmetry or reflexivity of `~@`: `T1 <@ T2`, `T2 <@ T1`, `T1 ~@ T2`.

Pointer data types are treated a bit differently. Pointers of types `T1P -> T1D` and `T2P -> T2D` are defined to be non-overlapping if either `T1P` and `T2P` are non-overlapping or if `T1P` and `T2P` are identical and `T2D` and `T2D` are non-overlapping.

[TBD: virtual containers]

2.16 Memory Channels

A ***memory channel*** is a mechanism for accessing a set of blocks in RAM that permits blocks to be announced substantially in advance of being accessed. Thus memory channels implement '***look ahead***' for memory accesses.

A memory channel implements a ***window***, which is a structured set of elements each associated with a member of some data set. Each window element contains a ***block descriptor*** that holds the address and length of the memory block that contains the data associated with the element. Block descriptors can also be marked as ***empty***, meaning there is no block to be accessed. The window has a ***reference point***, and window elements are addressed relative to this reference point. There are shift operations that move the reference point to a nearby window element.

Although we talk about blocks here, a block can be just a numeric array element, and can be as small as a single bit. Although we talk about each element of a memory channel window having its own block descriptor, an actual memory channel may use only block group descriptors, each of which functions as a group of more than one individual element block descriptor.

A memory channel is stored in a cluster. As such it is mostly an inline construction, though it can be passed to or returned from a function, and the function can be all or partly out-of-line.

The most common type of memory channel has a window that appears to be an array with **.dimensions**, **.lower_bound**`[`i`]`, and **.upper_bound**`[`i`]` being defined memory channel members. Such are called ***array windows***. If the memory channel cluster name is M, the window elements are referred to by M`[`i_0`,`i_1`,...]`, with M`[0,0,...]` being the ***reference point***.

The reference point can be shifted along any of the window's dimensions by the command M**.center**`[`i_0`,`i_1`,...]`, which shifts the window so that what was M`[`i_0`,`i_1`,...]` becomes M`[0,0,...]`.

Creating memory channels and completely reseting their reference points are specific to the type of memory channel, and are not covered in this section.

For most kinds of memory channels, block descriptors are computed automatically when channel is created, when the window is shifted, or when the data of a neighboring window element is arrives from memory. Immediately after a block descriptor is created, a read-ahead of the block is initiated. This read-ahead overlaps computation that does not use the block contents.

If a memory channel accesses arrays stored in memory, the channel block descriptors can be computed from the array coordinates of the reference point. Other memory channels use the contents of a block to compute the block descriptors of neighboring blocks in the window.

An example of the latter is a binary tree memory channel. Let M be such a channel, and let '`.L`' denote the left child of a binary tree element, '`.R`' the right child, and '`.P`' the parent. Then M`.L.R` denotes the right child of the left child of the reference point, M`.P.L` denotes the left child of the parent of the reference point, and M`.P.L.center` moves the reference point to this last element. The window of such a memory channel might contain the depth 2 subtree of the reference point plus that closest 4 ancestors of the reference point if these have been visited. When the reference point is moved, as soon as the reference point element has been read from memory, the descriptors for its children are built and the read of the children is initiated in parallel with other computation. When the children arrive from memory, the descriptors of their children are built and reads of the data pointed at are initiated.[5]

Some standard memory channel types are built into the L-Language. Others can be defined by users.

3 To Do

How can dynamically initialized locations be static.

Indirect address protocol.

[5]All this can actually be done with modern hardware: code is executed to read the reference point children and initiate the reads of their children, and a modern processor will automatically save the code that executes when a read of a reference point child completes and execute other code in parallel until the read does complete.

Functions.

Threads.

Function calls should be able to serve as lvalues so `x[i]=z` can update gc flags.

A Aliasing Hardware

The ultimate solution to the aliasing problem is new hardware. At its simplest, registers, which currently hold a datum, are replaced by triples of registers which hold a datum, an address, and selection codes. The register datum equals the value of the memory location at the register address. The selection codes determine which part of this memory location is read or written when the register is read or written. If any memory location is changed, the address of the location is checked against all the register addresses, and if any match, the corresponding register data are changed.

This is, however, not sufficient, because sometimes one register address is a function of another register's datum. For example, consider the unchecked code:

```
struct S { ...; int32 m; ... }
S * * x
S * *& y = * x
int32 *& z = y->m
```

If we consider `x`, `y`, and `z` to be registers, the address of `y` equals the value of `x`, and the address of `z` equals the value of `y` plus the offset of `m` in `S`.

If the value of `x` changes, this changes the address of `y`, which may change the datum of `y` and that may change the value of `y`. If the value of `y` changes, this changes the address of `z`, which may change the datum and value of `z`.

The way we accommodate this is to use the selection codes of `y` to specify that the address of `y` contains the value of `x` as an additive component, so that if the value of `x` is changed by adding Δ`x` then the address of `y` should be changed by adding Δ`x`. And similarly the selection codes of `z` specify that the address of `z` contains the value of `y` as an additive component.

So why should we bother with automatically updating additive inclusions of one value in the address of another value, and not bother with other expressions. The reason is that expressions such as '`(*x)->m`' are likely to be reused frequently in code (actually, in automatically generated code) and therefore need to be cached, whereas an expressions of the form '`c*d`'

will be reused comparatively rarely code and therefore are not worth special hardware.

Index