Ignite Language Overview

Peter Seymour

21 August 2008

Last updated: 2 March 2009

1 Introduction

Ignite is an entity-oriented programming language with a single built-in data type that supports static typing, type inference, polymorphism, operator overloading, type generation and concurrency. An entity is a generalisation of the traditional object in an object-oriented language and all programs are composed as a collection of entities.

Entities fall into different categories but are defined by a series of common features for instance member functions and variables. The purpose of an entity is to explicitly state how it is intended to be used and allow the language to aid its use. The singleton entity is a good example since the language runtime can ensure it is unique and prevent synchronisation and identity problems by prohibiting state.

This paper gives an overview of all of the language features paying specific attention to the use of entities. Many of the constructs should be familiar to anyone who uses object-oriented programming languages, however, the syntax is more explicit. It is hard to talk about some features without mentioning previously undefined features so the reader should be patient and return to unclear sections when more features have been introduced.

2 Types

Each entity has a statically determined type that falls into one of five categories. Each of these categories is handled separately. When a temporary variable is declared the compiler infers its type from the type of an assigned expression. Undefined variables are illegal so this type inference is always available. Member variables are explicitly typed.

2.1 Value

The unique built-in type `value` has instances from the non-negative integers modulo 256. Variables of this type are to be used for indexing, counting or constructing more complex types. The primitve operations available are

`+,-,*,/,%,<,>,<=,>=,==` and `!=`. Each has its usual meaning with the results being modulo 256. Here `%` is modulo arithmetic by the right-hand side. To declare a variable of value type use the keyword `val` to introduce the variable name. Examples of use are as follows:

```
val a := 21
val b := a
a := b + 1
```

All assignments are by-value which includes passing values to and from functions and operators and in messages. Note that none of the operations alter a variable in place, this can only be achieved by re-assignment. The `value` type also supports the prefix operator `$` which is an identity operator. This operator is intended to produce a `value` type from any type for use in conditional operators and indexing. So the following leaves the value of `a` unchanged.

```
val a := 21
a := $a
```

2.2 (Passive) Objects

Object entities are defined by the user to define new synchronous types. The features available depend on the sub-category of the entity being defined. Instances are constructed on the heap by the `@` (allocation operator) and variables assigned a reference to them. Here variables are introduced with the keyword `obj`.

```
obj a := @Foo()
obj b := a

val v := b.add( 21 )
```

Assignments including to and from functions and operators are by reference only. This is not the case in messages where deep copies of the arguments are made (see §**??**). At runtime all entities of this category will be similarly represented with zero or more state variables and a function/operator table for dynamic dispatch. Member function calls occur synchronously i.e. control is transferred into the target function. Where sub-typing is used assignments can occur to variables of the same type or a super-type. Type inference uses the exact type of the right-hand side to determine the type of variable declarations. These entities are analogous to classes in popular object-oriented languages.

2.3 Active Objects

Active objects are asynchronous versions of passive objects. The word passive is used to disambiguate regular objects from their active counterparts. Like passive objects they are constructed on the heap and have zero or more state

variables. However, they run as their own process and have a dedicated message queue. They do not support synchronous function/operator calls but allow asynchronous messages to be passed to them with arguments. All the arguments are deep copied first and the completed message added to the queue. Message handlers are declared in an analogous way to member functions and have a name along with typed arguments. Instances of active objects are assigned to variables declared by the keyword `act`. In all other ways they are like passive objects including the ability to be part of a type hierarchy.

```
act a := @Server()
act b := a

b..send( 21 )
```

2.4 Member Functions

Member function entities enscapsulate an object member function for deferred invocation. They are automatically generated whenever an instance is requested in the source and allocated to the heap. Each function entity has a single member operator `()` which accepts arguments of the same type as the target function and possibly a return type. Variables of this type are introduced by the keyword `fun`. For instance to create a deferred invocation of a function `add` on an object `Foo` taking one value argument and returning another would look like:

```
obj x := @Foo()
fun f := @x.add

val v := f( 21 ) / equivalent to f.( 21 )
```

Overloaded functions must be disambiguated by providing the argument types (but not the return type), in this case by `@x.f(:value)`. Operators can also be enscapsulated using their verbose syntax (see operator tables) such as in `@x.+(:Foo)`. The instance `x` is bound to the function entity as if it had been passed to the constructor of an object. The type of this entity is `Foo.(:value)->value`. Note that all functions are member functions so this method can be universally applied. A function object is a purely syntactic construction that could be programmer-supplied as a passive object.

2.5 Messages

A message entity is analogous to a member function but for active objects. The type syntax is slightly modified and no return types appear since they are not applicable for messages. An example shows this best.

```
act x := @Server()
mes m := @x..send
```

```
m..( 21 )
```

The type in this case it `Server..(:value)`.

3 Features

There are five primary types of feature: state, constructors, functions (and related operators), message handlers and sub-typing. Using combinations of these enables complex entities to be defined that relate to programming concepts. Each is discussed below.

3.1 Variables

An entity can declare member variables either as an individual, a sequence or a tuple.

3.1.1 Individual

Using one of the following statement forms depending on the category of the type a single variable is declared.

```
val v : value
obj o : Foo
act s : Server
fun f : Foo.(:value) -> value
mes m : Server..(:value)
```

These will be initialised with an entity in a constructor before first use but can be re-assigned at any later time.

3.1.2 Homogeneous Sequence

Between 1 and 256 variables may be simultaneously declared using the following syntax.

```
seq vs # [0...7 : value]
seq xs # [0...maxi : Foo]
```

In the first instance 8 variables are declared whereas in the second case `maxi+1` are declared. This is a compile-time construction and simply informs the compiler to declare a sequence of variables in place. More specifically it does not declare a new type just a sequence of variables of a previously defined type. The sequence is passed component at a time to another sequence declaration. For instance it is not possible to construct a sequence of sequences. The maximum index is specified as a `value` that is known at compile time either as a constant or a generator parameter of type `value` (see §**??**). To use one of these variables

the index argument will have the `$` operator invoked first. This ensures the index argument is of type `value` and enables user-defined types to be used.

```
val v = vs#4
obj o = xs#v
```

The index is taken modulo the length of the declared sequence (known at compile time) so that no error may occur. In addition to the component-wise use the sequence can be used as a whole with a simple syntax to construct the righthand side. This simply informs the compiler to generate a sequence of assignments, however, it lends itself to a situation where the length is unknown.

```
seq vs = [ 1, 2, 3 ]
seq ws = vs
seq xs = [ @Foo(), @Foo() ]
seq ys = [ @Foo()#0...maxi ]
```

In particular these sequences may be passed to and from functions, operators and to constructors. The inferred type of each sequence component is the most specific common super-type. If this does not exist then the sequence is rejected. Note in the last example the sequence contains `maxi+1` copies of a reference to the allocated entity. When specifying a sequence as a return type it takes the form `[0...7 :  value]`.

3.1.3 Heterogeneous Tuples

Like sequences, tuples allow for between 1 and 256 variables to be simultaneously declared. However, there is no restriction forcing the component types to all be the same. This means that runtime selection is prohibited and all indexing must be by a compile time constant `value`.

These examples show the syntax.

```
tup p # {:value,:value}
tup q # {:Foo,:value,:Foo}

val v = p#0
obj o = q#2
val w = q#1

tup p1 = { 1, 2 }
tup p2 = p1
obj f = @Foo()
tup q1 = { f, 7, f }
```

To specifiy a tuple as a return type use the syntax: `{:Foo,:value,:Foo}`.

3.2 Constructors

A constructor is required to put an entity into a known state after allocation on the heap using the built-in operator `@`. A constructor is an anonymous function introduced with the `constructor` keyword which may have arguments declared either implicitly or explicitly. For explicitly declared constructors there will be a sequence of initialiser arguments declared in the same order as their respective member variables. These will be either `value` constants, one of the arguments or `self`. They can be used individually or aggregated into sequence or tuple form. Should it not be possible to initialise a member this early on it can be deferred by placing a `?` in its place. With the exception of the sendable constructor a statement body must be present. Any members that have deferred initialisation must appear on the left of an assignment as their first use and must be used at least once.

```
val _x : value
val _y : value
tup _p # {:value,:value}

constructor ( x:value )
  @( x, ?, {x,x} )
{
  _y := _x + 1
}
```

The compiler can generate a simple constructor which simply takes an argument list equal to all the member variables. This constructor then initialises the arguments to the member variables.

```
constructor simple
{
}
```

In a similar manner there are three other implicit constructors: `shallowcopy`, `deepcopy` and `sendable` They all take an argument of the declared entity. The first performs a shallow copy by making an assignment to each member variable from the corresponding member of the argument. The statement body then executes. The second performs a deep copy by making assignments of deep copies of each corresponding member variable. This creates a chain of calls. Should any member fail to have a deep copy constructor then such a constructor cannot be declared for this entity. The third makes deep copies as in the second but does not permit a statement body to follow. The compiler inserts calls to this constructor to generate deep copies of all message arguments. Should any member fail to have a sendable constructor then such a constructor cannot be declared for this entity. By not having a statement body it is guaranteed to terminate with a completely distinct entity. These copy constructors are invoked using a more explicit syntax as shown in the following example:

```
obj/concrete ListNode
{
  val _v : value
  obj _next: ListNode

  constructor ( v : value )
      @( v, self )
  {
  }

  constructor simple
  {
  }

  constructor shallowcopy
  {
  }

  constructor deepcopy
  {
  }

  constructor sendable
}

obj x1 = @ListNode( 21 )
obj x2 = @ListNode( 29, c1 )
obj x3 = @ListNode[shallowcopy]( x1 )
obj x4 = @ListNode[deepcopy]( x1 )
obj x5 = @ListNode[sendable]( x1 )
```

There is no exception to these rules when using member variable sequences as in the following.

```
obj/concrete Pair
{
  val _xs # [0...1 : value]

  constructor simple
  {
  }
}
```

This can be used by `@Pair([1,2])`.

3.3 Functions and Operators

An entity can declare and potentially define member functions and/or operators. Operators only differ from member functions syntactically. In place of a function name is an operator symbol. This determines the operator precedence, number of arguments and position so that a formula can be reduced to a series of member function invocations. The positions are either `infix`, `prefix` or `postfix`. Operators are declared by one of these positions. If a defintion (sequence of statements) is not given then a sub-type must define it. All invocations are polymorphic namely the most specific function is used. To declare a function the following syntax is used.

```
function test( v:value, f:Foo ) -> Server
```

Notice that any type can be used as an argument or return type in either individual, sequence or tuple form.

```
function test( v:value, fs[0...7]:Foo ) -> {:value,:Server}
```

The return type can be omitted for pure procedures.

```
function reset()
```

The available operators are described in the following tables. The binding rules are that first postfix operators are applied (along with usual member function invocations) then prefix operators. Finally these results are made available to the infix operators which have their own precedence ordering as indicated. Parentheses may be used to change the order of evaluation. All infix operators are left associative.

Table 1: Prefix Operators

Name	symbol	verbose
plus	`+`	`.+()`
minus	`-`	`.-()`
value	`$`	`.$()`
not	`!`	`.!()`

Table 2: Postfix Operators

Name	symbol	verbose
apply	`(<args>)`	`.(<args>)`
query	`?`	`.?()`
prime	`'`	`.'()`

Table 3: Infix Operators

Name	symbol	prec	verbose
power	^	1	.^(<arg>)
multiply	*	2	.*(<arg>)
divide	/	2	./(<arg>)
modulo	%	2	.%(<arg>)
add	+	3	.+(<arg>)
subtract	-	3	.-(<arg>)
less than	<	4	.<(<arg>)
more than	>	4	.>(<arg>)
less than equal	<=	4	.<=(<arg>)
more than equal	>=	4	.>=(<arg>)
equal	==	5	.==(<arg>)
not equal	!=	5	.!=(<arg>)
equivalent	~	5	.~(<arg>)
and	&	6	.&(<arg>)
or	|	7	.|(<arg>)
choose	:	8	.:(<arg>)
right	->	9	.->(<arg>)
left	<-	9	.<-(<arg>)
value assign	=	10	.=(<arg>)
add to	+=	10	.+=(<arg>)
subtract from	-=	10	.-=(<arg>)
multiply to	*=	10	.*=(<arg>)
divide by	/=	10	./=(<arg>)
modulo by	%=	10	.%=(<arg>)

To provide an operator definition it must match the pre-described pattern. Where a definition is given this is supplied below in curly braces. Examples are given for each type:

```
postfix ' -> Foo
postfix () -> value
postfix ( v:value ) -> value
prefix - -> Foo
infix + rhs:Foo -> Foo
```

3.4 Message Handlers

For an active object message handlers are supplied in a similar way to functions but using the keyword `handler`. No return entity can be specified. Operators are not permitted as message handlers. For instance:

```
handler send( v:value )
handler reset()
```

3.5 Interface Implementation

A passive or active object may implement an abstract interface of the same type category (active or passive). When permitted it must define all undefined functions and operators in the interface (or message handlers in the case of active objects). This definition must agree with the arguments and possible return type. For an argument it must be either the type specified or a sub-type. The return type must be either the same type or a super-type. Where sequences are used the same rules apply for each component individually since a sequence is not a type.

For example if B is a sub-type of A then:

```
function f( b:B ) -> A
```

can be defined by

```
function f( a:A ) -> B
```

3.6 Sendable

To allow safe passing of arguments to an active object the environment must make a copy of all arguments. The value type can naturally be copied and so is intrinsically sendable. All active objects can be passed by reference since they are first-class concurrent objects. The same is true of message entities since they only contain a reference to an active object and all the arguments will have been copied in a sendable way. The only restriction remains with passive objects and function entities which must be deepcopied before being sent. This is achieved by calling the sendable constructor on each argument which in turn calls sendable constructors on their members. The sendable property is not strictly necessary since the enviroment can copy arbitrary graphs, however, it is desirable for the programmer to be able to control how much copying occurs when messages are sent.

4 Feature Order

Features in entities must be ordered such that declarations preceed all uses. This allows the entity to be compiled in one pass and adds some structure to the layout. For variables this implies they are listed before all functions, operators or message handlers that may use them. Some functions, operators or message handlers maynot be totally ordered i.e. they refer to themselves in a cycle. For this reason they can be forward declared and then defined later. The definition must match the declaration. For instance:

```
function g() / forward declaration

function f()
{
```

```
  g()
}

function g()
{
  f()
}
```

Within a definition the function, operator or message handler is declared even though undefined, however, this declaration is sufficient for recursion.

5 Entities

Entities are the top level unit for specifying a system. All the user-specified entities fall into either the passive or active category.

All entities have a type name that is prefixed by a path leading to it. For example `math.functions.Cos`. Where an entity needs to refer to another entity it must import the name to bring it into scope, `import math.functions.Cos`. Once imported it can be referred by its non-prefixed name e.g. `Cos`. Where names clash they can be renamed for that scope, `import math.functions.Cos as MCos`. The import mechanism defines a directed graph structure on the entities which can be searched recursively to ensure all dependencies are compiled in the correct order. To ensure this mechanism terminates the graph must contain no cycles. By a careful use of interfaces this prevents no problem for parent-child relationships. Each entity encapsulates a particular desire on the part of the programmer to represent a concept. This in turn creates constraints on which features are available to allow the entity to behave accordingly. For instance the singleton may not have state so it can be used safely across a distributed system.

To declare an entity first specify its category (`obj` or `act`) followed by a sub-category separated by a forward slash. The name of the entity follows optionally followed by a super-type. For instance:

```
obj/concrete Foo : Base
```

Each entity type is now briefly discussed and the traits of each type are summarised in a table.

5.1 (Passive) Objects

5.1.1 Concrete

Concrete objects define all functions and operators declared and must define all those from any super-type (i.e. an interface). They may have member variables and any type of constructor so can be sendable. These represent the most common type to be defined. Once allocated a constructor is called to initialise all member variables so at least one constructor must be defined.

```
obj/concrete Counter
{
  val _v:value

  constructor simple
  {
  }

  constructor ()
      @( 0 )
  {
  }

  function increment()
  {
    _v := _v + 1
  }

  prefix $ -> value
  {
    return _v
  }
}
```

This is then used by:

```
obj x := @Counter()
x.increment()
x.increment()

obj y := @Counter( $x + 7 )
obj z := @Counter( x )
```

5.1.2 Interface

An interface declares functions or operators but does not define them. An interface can inherit all the functions and operator declarations from another interface. Naturally an interface has no constructors or member variables and cannot be allocated directly onto the heap. This is a fundamental component of polymorphic design.

```
obj/interface Matrix : MathObject
{
  infix + -> Matrix
  infix - -> Matrix
  infix * -> Matrix
```

```
  prefix - -> Matrix

  function width() -> value
  function height() -> value
  function invert()
}
```

5.1.3 Singleton

A singleton object is a special concrete object that has no state (i.e. no member variables) but may implement an interface and is sendable. Since it is concrete it must define all functions and operators but does not have constructors. The environment creates an instance which is shared globally. This instance is referred to by the entity name prefixed by '`#`'. A singleton can serve a variety of purposes. It may represent a collection of functions in a sense acting as a namespace.

```
obj/singleton Math
{
  function sin( x:Decimal ) -> Decimal
  function cos( x:Decimal ) -> Decimal
}
```

Alternatively it could encapsulate a function and provide extra details.

```
obj/singleton Cos
{
  postfix ( x:Decimal ) -> Decimal

  function precision() -> Decimal
}
```

One way of using this singleton is `#Cos( d )` for some `Decimal d`. A final use might be to pass around information that is genuinely unique such as traits. Given a traits interface `EntityTraits` the following might be useful:

```
obj/singleton SingletonTraits : EntityTraits
{
  function allows_constructors() -> Boolean
  function is_passive() -> Boolean

  / ...
}
```

Then this could be passed around by `set_traits( #SingletonTraits )`.

A singleton is sendable since it has no state. For this reason it is logically unique across a distributed system even if there exist multiple instances of it.

5.1.4 Native

Native objects are passive singletons except that only functions are permitted whose definitions are not read by the compiler. When a native entity is parsed the function bodies are copied to their target without modification. The target platform of a native entity is specified as its subcategory. These objects are implementation dependent but encapsulate a collection of externally defined functions. They are for use by library implementers.

```
native/win32 IO
{
  function read() -> char
  {
    /...
  }
}
```

5.2 Active Objects

5.2.1 Concrete

These are defined similarly to their passive counterparts but with message handlers replacing functions and operators. Interface implementation refers to implementing active interfaces. Constructors are declared as for concrete passive objects but the sendable constructor is unnecessary. However, constructors behave like message handlers. When an active entity is declared as allocated its constructor arguments are copied just like any other message handler. Once it is actually allocated (possibly at some other location in a distributed system) the constructor message will be the first in the queue. It is handled like any other message. An active entity will continue indefinitely unless terminated by an escaping exception (see later sections).

5.2.2 Interface

Again analogous to passive interfaces except message handlers replace functions and operators. Like active concrete objects the sendable property is not relevant.

5.2.3 Singleton

Unlike passive singletons these are permitted to have member variables. They must also define a no argument constructor. The environment constructs active singletons onto the heap in an arbitrary order if at all. They may implement an active interface. These entities represent parallel processes in a server model. Like concrete active objects they start processing messages as soon as they are allocated. A constructor message will be the first in the queue.

```
import ProcessManager
```

```
act/singleton FileManager
{
  val _open_handles:value

  constructor ()
    @( 0 )
  {
    #ProcessManager..register( self )
  }
}
```

5.3 Summary Of Entity Traits

These tables summarise the traits of each entity type in terms of the restrictions on their features.

Table 4: Passive Entity Traits

category	obj		
sub-category	concrete	interface	singleton
sendable	yes	n/a	default
inheritence	sub	super/sub	sub
member variables	yes	no	no
constructor (anonymous)	yes	no	no
constructor simple	yes	no	no
constructor shallowcopy	yes	no	no
constructor deepcopy	yes	no	no
constructor sendable	yes	no	no
operators/functions	defined	declared	defined
message handlers	no	no	no

6 Generators

Generators are used to generate new entities and features by supplying a series of static type and value parameters. Other names for this method include generics, type parameterisation and templates. The generator definition takes parameter types according to the type categories. It then infers by use what features are required and uses these as constraints on which types may be supplied. A simple example is a typed pair:

```
generator< T1:obj, T2:obj >
obj/concrete Pair
{
  obj _first:T1
```

Table 5: Active Entity Traits

category	act		
sub-category	concrete	interface	singleton
sendable	default	n/a	default
inheritence	sub	super/sub	sub
member variables	yes	no	yes
constructor (anonymous)	yes	no	no-arg
constructor simple	yes	no	no
constructor shallowcopy	yes	no	no
constructor deepcopy	yes	no	no
constructor sendable	no	no	no
operators/ functions	no	no	no
message handlers	defined	declared	defined

```
  obj _second:T2

  constructor simple
  {
  }

  function first() -> T1
  {
    return _first
  }

  function second() -> T2
  {
    return _second
  }
}
```

A generated entity is used by supplying type parameters at construction time.

```
obj pair := @Pair<Foo,Bar>( foo, bar )
```

Each generated entity is a distinct and genuine entity no different from its hand-coded version. The generator mechanism simply fills in all the details. The use of one of the type parameters in a function say may restrict its domain. If for example a certain member function is used then that type must support it.

```
generator< T:obj >
obj/concrete LazyAdd
```

```
{
  obj _lhs:T
  obj _rhs:T

  constructor simple
  {
  }

  postix () -> T
  {
    return _lhs + _rhs
  }
}
```

In this example the type `T` must be an object entity that supports the addition operator i.e.

```
postfix + rhs:T -> T
```

Where a generator parameter is an active object the same rules apply except handlers are a constraint rather than functions or operators. If a value parameter is supplied this is then a constant.

```
generator< max_index:val, T:obj >
obj/concrete Array
{
  obj _items # [ 0...max_index : T ]

  constructor ( t : T )
    @( [ t#0...max_index ] )
  {
  }

  constructor simple
  {
  }
}
```

To generate such an entity requires a constant value, `@Array<100,Foo>(@Foo())`. Or this can inferred by the simple constructor, `@Array([@Foo(),@Foo()])`.

The second use of a generator is for functions, operators and message handlers. The generator syntax is supplied directly before the operation to be generated and again supplies type parameters or constant values.

```
obj/singleton Factory
{
  generator< T:obj >
```

```
  function make() -> T
  {
    /...
  }
}
```

Note that these generators may not be used for undefined operations i.e. within interfaces. However, interfaces may themselves be generated.

A compiler need not duplicate code to produce these entities since all invocations are by dynamic dispatch so only the relevant dispatch tables need to be altered. For instance if a type parameter `T` is used then supplying a `Foo` in its place only requires a mapping from the operations of `T` to their definitions in `Foo`. The generated entity can keep a copy of this new dispatch table so each generation only adds a dispatch table for each of its type parameters. All the other code is shared.

7 Exceptions

Exceptions are the primary form of error handling. At any time an active or passive object may be thrown as an exception. The call stack then unwinds until a suitable handler is found and executed. If no handler is found then the exception will finally reach a message handler at which point the active object owning that handler will terminate. Should further messages be sent to a terminated active object then the active object will be thrown in the scope of the caller. See the section on concurrency for why this is so.

8 Null Entities

The language supplies generated (singleton) null sub-types for all passive and active objects. These can be assigned to references of the base entity type or any of it super-types. Each function and operator of a passive object is simply defined as `throw self`. In the case of an active object the null sub-type is never started so attempts to send a message to it cause it to be thrown in the scope of the caller. This is analogous to an exception escaping a passive null type into the caller scope. Since they are singletons they may be accessed with the `#` prefix. Their full type name is the base entity's type name prefixed by `!`. For instance if there is a passive object `Foo` with a function `test` the following is valid.

```
obj f := #!Foo
f.test()        /will throw f
```

The ugly syntax is to discourage use since they are provided for library implementers.

9 Statements

To define a function, operator or message handler requires a statement block. A block is a sequence of statements enclosed in curly braces. These statements come in a small number of forms.

9.1 Declarations

To declare (and define) a new variable for use in local scope use one of the type categories as a keyword followed by the variable name and assign it a right-hand side expression.

```
val v := 21

obj Foo f := @Foo()
fun fn := @f.count

act s := @Server()
mes m := @s.send
```

9.2 Expressions

These are the usual right-hand side elements of a statement that may return a value. No left-hand side is required even if an entity is returned. An expression is built from function or operator invocations, message sends or heap allocations. They may be chained together as their types permit.

```
obj m1 := @Matrix<4,4>()
obj m2 := m1.invert().transpose() + m1
```

Operator precendence determines how this resolves itself into a sequence of invocations. An operator can be explicitly called as a function by using the `.` syntax.

```
obj m2 := m1.+( m1 )
```

9.3 Conditionals

A conditional statement makes a test on an argument by invoking the `$` operator and comparing the value returned with 0. This permits user-defined types to be used within conditional statements. Following the test must be a statement block.

9.3.1 if

The `if` statement executes the statement block if the test value does not equal 0, otherwise it skips it.

```
if test /compare $test with 0
{
  /...
}
```

9.3.2 while

The `while` statement executes the statement block repeatedly while the test value does not equal 0, otherwise it skips it.

```
while test /compare $test with 0
{
  /...
}
```

9.3.3 for in

The `for in` statement introduces a new passive entity reference that is not available outside of its scope. This reference is used for testing as in a `while` statement. Note that `$` will be invoked on this reference before each interation.

```
for x in list.items() /compare $x with 0
{
  /...
}
```

This is equivalent to:

```
obj x := list.items()
while x
{
  /...
}
```

9.4 return

To return an entity from a function or operator the `return` keyword takes an expression argument and returns that. To exit a function or operator without returning an entity just use `return` with no expression.

```
function add( v1:value, v2:value ) -> value
{
  return v1 + v2
}
```

9.5 yield

Analogous to the no argument `return` of functions and operators is the `yield` statement for message handlers. Once executed the current message handler exits and the next message can be processed from the queue.

```
handler reset()
{
  _count := 0
  yield
}
```

9.6 throw

Analogous to returning an entity. To throw an exception entity use the `throw` keyword and the result of the expression will be thrown.

```
throw @FileNotFound( path )
```

9.7 try/catch

A `try` block is a statement block such that any exception thrown during its execution may be caught by a proceeding `catch` block. The `catch` statement requires an argument which if matched against the thrown entity will bind to the entity as in a function call. The statement block of the catch is then executed. There may be multiple `catch` blocks.

```
try
{
  #FileManager.open_file( path )
}
catch( e:FileNotFound )
{
  /...
}
catch( e:FilePermissionDenied )
{
  /...
}
```

Any uncaught exception escapes to the block enclosing the `try` block.

10 Concurrency

Each active object runs independently of the others although the definition of message handlers can make use of synchronous calls to passive objects. This is

a powerful mechanism to write programs in a traditional style but build higher-level constructs as processes using active objects. Each active object is a process and cannot alter any other except by passing a message. The object graphs are kept distinct so the computations may exist anywhere. If any one process gets stuck it will not affect the rest. Where processes are to be simulated then resources need to be fairly distributed. This might mean allowing each active object to have a certain number of steps before switching to another. A round-robin allotment would ensure no process can starve another. Active objects with no pending messages that have yielded can be put to sleep.

A path of execution always starts by an active object handling its initial construcor message. It then proceeds synchronously through the passive objects forever or until it eventually yields or an exception escapes the handler forcing it to terminate.

11 Garbage Collection

So far a lot has been said about constructing entities and manipulating them, however, at some point entities cease to be referenced. This policy provides an infinite memory model which is the intent of the language. These redundant entities can be safely garbage collected at anytime by the runtime environment without the application being semantically affected. The only effect may be a loss of performance but this is an implementation issue outside of the scope of the language definition.

12 Syntax Extensions

When creating an object instance it maybe cumbersome to provide constructor arguments using only `value` constants. For instance numbers greater than 255 or string literals would be fiddly. This section proposes a few extensions at the syntax level to remedy these problems.

12.1 Numbers

Any sequence of decimal digits can be read as their base 256 expansion with each coefficient being taken as a single `value` constant. The most significant `value` coming first and all leading zeroes stripped away (except for zero itself). For instance 1000 would be expanded as `[3,232]`. Any constant less than 256 will not be expanded.

This can be extended to non-decimal bases. For hexadecimal write `0x123456` as `[18,52,86]`. Or in binary write `0%1111000010101010` as `[240,170]`.

For negative numbers a zero can be inserted at the front since all non-negative numbers have zeroes removed. In this case `-1000` would be writen as `[0,3,232]`. This introduces a distinction between `0` and `-0`.

12.2 String Literals

In a similar manner using a character set of no more than 256 characters enables string literals to be converted to sequences of values. One for each character. So `"hello"` would be interpreted as `[104,101,108,108,111]` in ASCII.