

# Flame: An Intermediate Representation for Ignite Entities

Peter Seymour

10 September 2008

Last updated: 19 December 2009

## 1 Introduction

The flame acronym stands for Full Listing of A Modified Entity. It specifies partially compiled entities from the ignite language, see [1]. Since the ignite language is intended to be parsed in one pass the flame specification should also be read once from top to bottom. In this sense it is a modified version of the original entity definition and is referred to simply as a *flame*. The compilation process is primarily to compile statement blocks, type check and resolve types into indices. Other information about the entity is purposefully maintained in its original form.

An ignite script depends on various other entities and then defines an entity or entity generator. Once this has been compiled any generator parameters have had their requirements implicitly specified so these required types can also be listed. A flame is then a sequence of types either imported, defined or required. All references to a type are specified as an index into the ordered list that defines the flame. The value type is implicitly the 0<sup>th</sup> type in each flame. Every function, operator or message handler is compiled to a sequence of instructions for a virtual machine.

A runtime environment then loads each flame and all of their dependencies so that it can construct the runtime entities. There is an element of runtime complexity for generated types as each generator can give rise to multiple types. Note that generated types share the same instruction blocks with all entities from the same generator. The runtime environment may interpret or further compile the instructions. This is sufficient for system execution.

At compile time a dependent ignite script can be loaded directly and compiled or its flame form referred to. This second situation imposes the condition that a flame must retain all external typing and naming. Therefore all function, operator and message handler names, named arguments and named generator parameters must be preserved. The programmer-supplied names are optional but aid resolution of compile and link errors as well as documenting compiled libraries. All member variables and instructions are private to the entity and need not retain this information.

## 2 Modified Entity Format

This paper uses a syntax that assumes a flame in text form, however, other storage schemes are not excluded such as binary, in memory etc. Spacing is significant in that a space acting as a delimiter is expected to be a single space character. Indentation is used to group elements hierarchically into levels. A sequence of tab characters are used at the beginning of a line to place it on the required level.

## 2.1 Linked Types

Types specified in other flames are linked to the current entity by their full names with the `lnk` keyword. An optional hash value of the linked flame can be provided after a '#' character.

```
lnk math.Cos
lnk int#45de0a1f
```

The runtime can ensure that the correct version of the dependent type is being used from its hash value. If the hash value is not present then the version is assumed to be correct.

## 2.2 Generated Types

When a generator is used it generates a complete entity. These are declared in any flame that depends on one by supplying the type parameters. The generator flame will already have been linked in a preceding section. These generated types may appear in multiple flames but should be uniquely represented. The format is specified by the `gen` keyword followed by the generator type index and the indices of all the type parameters or `value` constants prefixed by a '#' character.

```
gen 2<1,3,#127>
```

## 2.3 Implied Types

Implied types specify precisely what features are required by any generator parameter. The compiler will have determined these and they are then listed here. When an entity is generated first a check is made that it has all the required features. The runtime then can construct an augmented dispatch table that maps the required functions, operators and handlers to their specific implementations in the target entity. An invocation on an implied type then uses the entry in the caller entity's dispatch table rather than the callee's table. This is known as a cross-call.

A double-cross-call uses a dispatch table referenced on the top of the stack. This can be used in generated functions that pass a dispatch table as a runtime argument. This prevents boxed arguments needing to be allocated for each invocation.

An implied type is introduced by the `imp` keyword followed by the parameter name as it appeared in the original source code and its category (`obj`, `act`, `fun` or `mes`). This name is for external reference only since it will be used according to its index in the *flame*. In the case of `fun` an apply operator will be required but for `mes` an un-named message handler should appear. At the next level (one tab in) are a list of all the function, operator and handler signatures. A signature retains argument names as they appear in the source code but types are listed by their index in the current context. Spaces are not permitted in the argument list.

```
imp T:obj
  function f(v:0)
  operator + rhs:2 -> 2
  prefix - -> 2
```

## 2.4 Implied Values

Where a generator parameter is a `value` constant it is given a type index as an implied type is. No features need to be declared since the built-in type is pre-determined. Again the name is for external reference only.

```
imp n:val
```

## 2.5 Aggregated Types

For sequences and tuples these are forward declared to admit them an index. A sequence requires the type index of the component type and its length. This length is either a constant or an implied value index. A tuple requires a list of all component type indices.

```
seq 7[#128]
seq 9[27]
tup 1,7,9
```

## 2.6 Defined Types

The actual entity being defined is defined in the same way as in the source code. For non-generator entities an empty generator clause is attached. In all other cases the generator parameters are listed as either the indices of implied types or values. Note the super-class is specified by its index. Once defined it can be given an external name and exported to be imported by other *flames*. Only one export may appear.

```
generator<2,8>
act/concrete :1
  handler send(v:0)
  RET

export Foo:17
```

Entities from the `fun` and `mes` categories which are compiler generated are also defined in this way.

```
fun
  rsv 2
  constructor simple
  RET

  postfix ()
  RET
```

### 2.6.1 Member Variables

The member variables do not need to be named but their types need to be known. Use the `rsv` keyword to reserve space for the member variables where the index specifies a previously declared type.

```
obj/concrete Foo
  rsv 3
  rsv 4
```

### 2.6.2 Functions, Operators and Handlers

The signatures of functions, operators and handlers have already been discussed. Since they are part of the public interface of the entity they are left largely unmodified by compilation. The same is not true of the statement blocks that may define them since this is the real purpose of compilation.

When a statement block is given it is a list of instructions for a virtual machine with all references and invocations resolved to the desired index. Any conditional control flow is also resolved to the number of instructions to be skipped. The next section details the instruction set.

Prior to the statement block is an entry to indicate the temporary local variables that are required. This is in the same form as for the member variables of an entity.

```
act/concrete Foo:1
  handler add(v:0)
    rsv 0
    rsv 16
    ...
```

### 2.6.3 Instructions

All instructions have a mnemonic and optionally a parameter. This parameter is either a constant of value type, an index, a type index or a jump offset. Instructions use a working stack to make intermediate calculations. The context of a function, operator or handler is defined by different variables that can be accessed. The self variable is a reference to the entity on which the operation was invoked. The locals are temporary storage allocated on the working stack for use in this invocation. The member variables are those for the self entity. The runtime will have constructed a series of permanent entities such as singletons. The variables are arranged so they can be referred to by index in the following order.

$$self, mem_1, \dots, mem_i, arg_1, \dots, arg_j, lcl_1, \dots, lcl_k$$

The instructions are divided up into groups of related behaviours.

## Load

Each of these pushes an item onto the top of the working stack.

### **LDDIR i** (Load Direct)

Load a copy of the **i'th** variable.

### **LDIND i** (Load Indirect)

Remove the top element of the stack as a value **j**. Load a copy of the **i'th** variable's **j'th** component.

### **LDVAL v** (Load Value)

Load a constant value **v** onto the stack.

### **LDENT t** (Load Entity)

Load a reference to a permanent entity onto the stack with type index **t**.

## Store

Each of these removes the top item from the stack and stores it elsewhere.

**STDIR i** (Store Direct)

Store and remove the top stack element to the **i**'th variable.

**STIND i** (Store Indirect)

Remove the top element of the stack as a value **j**. Store and remove the now top stack element to the **i**'th variable's **j**'th component.

**POP** (Pop)

Remove and discard the top element from the stack.

## Arithmetic

Each of these treat the top two items as values and replaces them with the result of a binary arithmetic operation. The top value being  $t$  and the next item  $s$ .

**VADD** (Value Addition)

Replace by  $s + t$ .

**VSUB** (Value Subtraction)

Replace by  $s - t$ .

**VMUL** (Value Multiply)

Replace by  $s \times t$ .

**VDIV** (Value Divide)

Replace by  $s/t$ .

**VMOD** (Value Modulo)

Replace by  $s \bmod t$ .

## Comparison

Each of these treat the top two items as values and replaces them with the result of a binary comparison. The top value being  $t$  and the next item  $s$ .

**VCL** (Value Compare Less Than)

Replace by 1 if  $s < t$ , otherwise by 0.

**VCG** (Value Compare Greater Than)

Replace by 1 if  $s > t$ , otherwise by 0.

**VCLE** (Value Compare Less Than Or Equal To)

Replace by 1 if  $s \leq t$ , otherwise by 0.

**VCGE** (Value Compare Greater Than Or Equal To)

Replace by 1 if  $s \geq t$ , otherwise by 0.

**VCE** (Value Compare Equal To)

Replace by 1 if  $s = t$ , otherwise by 0.

**VCNE** (Value Compare Not Equal To)

Replace by 1 if  $s \neq t$ , otherwise by 0.

## Invocation

These instructions invoke functions and operators or send messages. Operators at this level are indistinguishable from functions. When functions are invoked control is transferred to the statement block indicated in the relevant dispatch table. When messages are sent no control transfer is seen to take place.

**CALL i** (Call Function)

Call the **i'th** function in the dispatch table for entity on the top of the stack.

**CALLC i** (Cross-Call Function)

Call the **i'th** function in the dispatch table for the self entity. This is used in generated entities.

**CALLDC i** (Double-Cross-Call Function)

Call the **i'th** function in the dispatch table for the table referenced on the top of the stack. This is used for generated functions.

**SEND i** (Send A Message)

Send a message to the **i'th** handler of the active entity on the top of the stack. The descriptor for this handler will determine which arguments are to be copied and clean up the stack accordingly.

**RET** (Return)

Return from the current function to the instruction after the point of invocation. Stack clean-up is to be performed by the callee.

**RETE** (Return Expression)

Return from the current function to the instruction after the point of invocation with the top stack item left in place. Stack clean-up is to be performed by the callee.

**YIELD** (Yield)

Suspend this process temporarily to allow other processes to be run if necessary. This is a hint to the runtime.

**SIGNAL** (Signal)

Invoke the message object specified as a signal-channel (additional message handler argument) with object on top of the stack. Combined with a signal construction this implements the **signal** statement.

**TERMINATE** (Terminate)

Terminate the current process. Combined with **SIGNAL** this implements the **exit** statement.

## Control Flow

### **BEGIN** *o* (Block Begin)

Mark a block of instructions as beginning. An **END** or **ENDR** instruction must be located *o* instructions forward of here.

### **SKIPZ** *o* (Skip If Zero)

Skip forwards by *o* instructions if the top item on the stack is equivalent to value 0, otherwise proceed to the next instruction. An **END** or **ENDR** instruction must be located *o* instructions forward of here.

### **END** *o* (Block End)

Mark a block of instructions as ending. A **BEGIN** instruction must be located *o* instructions backwards of here.

### **ENDR** *o* (Block End Repeat)

Mark a block of instructions as repeating. A **BEGIN** instruction must be located *o* instructions backwards of here.

## Heap Control

### **ALLOC** *t* (Allocate)

Allocate an uninitialised entity on the heap with type index *t* and push a reference to it onto the top of the stack.

## References

[1] “Ignite Language Overview”, Peter Seymour.