The Raven System [1]

Donald Acton, Terry Coatta and Gerald Neufeld

Technical Report TR 92-15

August 31, 1992

Abstract

This report describes the distributed object-oriented system, Raven. Raven is both a distributed operating system as well as a programming language. The language will be familiar to C programmers in that it has many constructs similar to the C programming language. Raven uses a simple, uniform object model in which all entities, at least conceptually, are objects. The object model supports classes for implementation inheritance and type checking. Both static and dynamic typing as well as static and dynamic method binding is supported. Object behavior is defined by the class of the object. The language is compiled (rather than interpreted) and supports several features that improve performance. Support also exists for parallel and distributed computing as well as persistent data. Raven is designed specifically for high-performance parallel and distributed applications.

[1] This work was made possible by grants from the Natural Sciences and Engineering Research Council of Canada.

1 Introduction

The primary goal of Raven is to provide an object-oriented development environment for building high-performance distributed and parallel applications. Such an environment must be easy to use, making the task of building distributed and parallel applications relatively simple. Many design decisions were considered in accomplishing this goal. As a result, the Raven system was based on the C programming language primarily because many programmers are already familiar with C. Raven differs from other C-based languages in that it maintains, at least conceptually, a consistent object model. All items in Raven, including code and data, are objects. Special support in the language is provided for the built-in types `Int,` and `String`.

The Raven class system is similar to that of Objective C [Cox86], Smalltalk [GR83] or Calico [EFM+91], rather than C++ [Str86]. However, unlike Objective C and Smalltalk, Raven is statically typed. A form of dynamic typing can be accomplished by using the `Any` type. Method binding is normally done at runtime, resulting in dynamic method invocation. However, it is also possible to obtain static method binding, resulting in greater efficiency. Raven's class system is used both for implementation inheritance as well as type checking. Like Smalltalk and Calico, all variables are references to objects. Hence references are implied in the language.

Raven differs from many object-oriented programming languages in that notions of concurrency and distribution are part of the language (analogous to Concurrent Smalltalk). Raven incorporates the properties of persistence, concurrency, security, recovery and distribution orthogonally to the class definition. That is, a class can be defined independently of these properties. Properties are assigned to an instance of a class when the instance is first created. Any combination of these properties can be specified. Hence, for example, objects from the same class may be persistent, concurrent, secure and atomic or volatile, sequential, unsecure and nonatomic. The language also supports several novel constructs for concurrent and parallel object execution.

1.1 A Raven Example

It is likely easiest to introduce Raven by example. We assume that the reader is familiar with C.

Figure 1 presents the classic pencil cup example. The class `Weighable` is an abstract class used only for type checking. Via Raven's type compatability rules only objects whose class implements the weight method can be added to a pencil cup. The cup class is simply a

```
class Weighable {                           // define an abstract class for type checking
    behavior Int weight();
}

class Cup <- Object {
    Int               size;                 // number of items in pencil cup
    Array[Weighable]  cup;                  // array holding weighable cup elements

    constructor();
    behavior    Int       insert( Weighable item );
    behavior    Weighable remove();
    behavior    Int       empty();
    behavior    Int       weight();
}

constructor Cup {
    cup = Array::new(MAX_SIZE);
    size = 0;
}

behavior insert {
    if( size >= MAX_SIZE )
        return ERROR;
    else {
        cup::PutAt(item, size);
        size++;
        return OK;
    }
}

behavior remove {
    if( size == 0 ) return nil;
    size--;
    return cup::ItemAt(size);
}

behavior empty {
    if( size == 0 ) return true;
    else return false;
}

behavior weight {
    Int total_weight, i;
    total_weight = 0;
    for( i = 0; i<size; ++i )
        total_weight += cup::ItemAt(i)::weight();
    return total_weight;
}

// The following is a fragment of code that uses the pencil cup class.
// Pencil is a class type that is subtype of Weighable.

Cup    pencil_cup;
Pencil pencil;
pencil_cup = Cup::new();
if( pencil_cup::insert( pencil ) == ERROR )
    "Cup overflow"::print();
else {
    "Cup weight = "::print();
     pencil_cup::weight()::print();
}
```


Figure 1:

stack (implemented as an array of weighable objects) that inherits directly from the top class Object. The class description is introduced by the keyword `class`. The instance variables are next, followed by the method definitions. The behavior of each method is defined immediately following the class construct.

Note that the formal parameters are not repeated (i.e. they are defined in the interface but not in the implementation). This implies that the formal parameters must also include the parameter names as well as their types. The `constructor` method is a special method that is automatically invoked whenever an instance of the class is created. (Instance creation is done via the `new` method being invoked on the desired class.)

Object invocation is defined by the object identifier followed by "::", followed by the method name. Inside a method behavior, instance variables are referenced in a manner similar to method invocation.

2 Object model

Within Raven all entities (code and data) are objects. The term class defines the unit of abstraction for objects. Object behavior is encapsulated and preserved within a class. Only single inheritance is currently supported. The root of the class hierarchy is the class `Object`. Each class c is an instance of a meta class with the name $*c$. For example, class `Set` is an instance of class `*Set`. Meta classes are instances of class `**Class` which is an instance of itself. All classes (including the meta classes) inherit from the root `Object`. As such, it is possible to have arbitrary class variables or methods in the created classes.

Every class also has an (implicit) type, called the class type. The type name is the same as the class name. Hence when we defined class `Cup` in the previous example, a type with name `Cup` was also implicity created.

New instances or objects are created via the `new` method and removed via garbage collection. `new` is a method of `*Class` that cannot be overridden. Each class definition may specify a `constructor` method. The `constructor` method is implicitly called after the `new` method completes. Its role is to initialize the newly created object.

Objects can only be accessed via method invocations. Normally, instance variables are private and not directly accessible. It is possible, however, to indicate that an instance variable is public. In this case, the compiler automatically generates put/get methods for the public variable for dynamic binding. Access to public instance variables is via the appropriate method invocation. The appropriate method is implied by how the variable is accessed. If it is accessed on the left-hand-side of an assignment then the put method is used; if on the right-hand-side, the get method is used. If the class is defined as public, all its instance variables can be referenced externally. Also, an object's private instance variables can be directly accessed by any of its subclasses.

Classes are defined statically via the compiler and run-time system. There is currently no support for dynamically (at run-time) creating new classes or methods.

An object is referenced via a variable. All variables are typed. A variable of type `T` can reference any object whose class type is equal to or a subtype of `T`. The only permissible variables are references to objects. The language is therefore safe in the sense that no pointers exist. A variable always refers to a valid object. Variables are initially set to the `nil` object. The system has a builtin type called `Any`. A variable of type `Any` may reference an object of any class.

In Raven, a reference to an object is called a capability. As the name implies, a capability indicates that it has rights to access the behavior of an object. An object may have more than one capability, where each capability has different access rights. Access rights are on a per method basis. That is, a capability indicates whether or not it has the right to invoke the object's methods. The capability returned from the `new` method is called the primary capability. It contains the default access rights. Additional capabilities with different access rights can be created via the `alias` method.

2.1 Class Interface

In the simplist case, the interface of a class includes the complete class definition. This is the case in Figure 1. However, it is possible to separate the interface from the implementation. This is particularly useful in a distributed environment where the implementation of a class is not replicated at all of the machines. The interface is simply the class definition minus any implementation code. For instance, an interface to the `Cup` class is:

```
class Cup <- Object {
    constructor Cup();                  // constructor method
    behavior    Int insert(Any item);   // add an item to a cup
    behavior    Any remove();           // remove an item from a cup
    behavior    Int empty();            // check if cup is empty
    behavior    Int weight();           // determine weight of cup
}
```

This interface can be include via the preprocessor (using #include). Invocation can then be performed on any instance of `Cup`. It is required that either the interface or the entire class definition (interface and implementaion) be included in the compilation. This applies to all of the classes that are inherited as well.

It is also possible that several different interfaces to a class can exist. For instance, if only the method `weight` is to be visible to a user, we could export the following interface.

```
class Cup <- Object {
    behavior    Int weight();       // determine weight of cup
}
```

This would restrict the user to just the `weight` function. Many such interfaces to a class may be defined. It is also common to create classes that do not contain any behavior code. Such classes can never be instantiated and are only used for type checking. In Figure 1 for example, the class `Weighable` was defined. No code was given for the `weight` method. The sole funtion of this class was to only permit an object whose class implements the weight method to be added to a pencil cup object. Classes like `Weighable` can also use inheritance, however in such cases inheritance is used for interface inheritance rather then implementation inheritance.

2.2 Parameterized Classes

Raven also supports static type parameterization of classes. For example, consider the following parameterized array class with a sample of how it might be used:

```
class Array[X] {
  behaviour   X atGet(Int index);
  behaviour     atPut(X item, Int index);
  constructor(Int size);
}

// now a segment of code which makes use of the class

behaviour SomeBehaviour {
  Array[Int]    anIntArray;
  Array[String] aStringArray;
```

```
  anIntArray   = Array::new(10);
  aStringArray = Array::new(5);
  anIntArray::atPut(5, 0);
  aStringArray::atPut(``hello'', 2);
  aStringArray::atGet(2)::print();
}
```

The identifier `X` enclosed in brackets following the class name is a type parameter. Once it has appeared in the opening of the class definition, then it may be used as a legitimate type name throughout the definition of the class. For example, the `atGet` behaviour takes an integer index and returns the item stored at that index. The return type of this behaviour is declared to be `X`. Later in the example, the `atGet` behaviour is invoked on the variable `aStringArray`, which is declared as having type `Array[String]`. The declaration establishes a binding between the type parameter `X` and the concrete type `String` for this variable. The return type of `atGet` when invoked on `aStringArray` is, thus, `String`.

Programming languages such as C have traditionally provided a certain number of built-in parameterized types. Facilities for programmer-defined parameterized classes are often quite weak (e.g. the use of `void *` in C to implement generic data types). In Raven, there are no built-in parameterized types having special properties. The programmer has complete access to the facilities required to create parameterized types. Currently, Raven does not support constrained type parameterization.

2.3 Distribution and parallel objects

The example in Figure 1 shows a typical sequential object as one might expect to see in C++ or Objective C. The system also supports access to objects distributed across the network, parallel threads of control through objects and concurrent object invocation. The underlying design goal in all of this support is transparency.

Within Raven, transparency is viewed both as static and dynamic. By static transparency we mean that the programmer can write object specifications with no knowledge of whether the object will be accessed locally or remotely. By dynamic transparency we mean that object invocation is identical for local and remote objects.

In Raven it is possible to create multiple threads of control. Each thread can access any object. If more than one thread accesses the same object, then concurrency control is used to coordinate access. It is also possible to suspend execution inside an object and have a different thread resume the suspended execution (thread) at a later time.

The normal model for object invocation is that the thread that invokes a method blocks until the method returns control. It is, however, possible to start an invocation and some time in the future to access the result, while continuing to execute. Using this mechanism it is possible to invoke several methods in parallel.

2.4 Object Properties

Each object is assigned a set of system properties. These properties define how the object is managed by the system. Properties are divided into five categories: storage, concurrency, security, recovery and distribution. Properties can be specified statically as part of the class definition or dynamically when an instance of a class is created.

The storage category includes the properties persistent and volatile. If an object is persistent, the Raven object store makes sure that the object is written to disk after any invocation modifies the object's instance data. Whether an object is persistent or volatile is determined at the time the object is created.

The concurrency category includes the properties controlled and uncontrolled. These properties indicates the system support for concurrent access to the object's instance data. If the object is controlled then the system makes sure that multiple threads accessing the object are subject to concurrency control, otherwise no control is implied. In this situation it is the programmer's responsibility to provide concurrency control as required.

The security category consists of the properties restricted and unrestricted. If an object is unrestricted, then a capability for that object may access any of the object's behaviors. If the object is restricted, then a capability may only have restricted access to the object. That is, only a subset of the behaviors may be permitted.

The recovery category consists of the properties atomic and nonatomic. If the object has the atomic property, its behavior maintains the characteristics of atomic transactions such as "all-or-nothing" and "isolation".

Finally, the distribution category indicates how an object is maintained. It includes the properties; local, migrate, and replicate. If an object is local then it is only resides within its local domain (address space or machine). Local objects may be referenced remotely. The migrate property indicates that the object may move from domain to domain. While replication indicates that the object is replicated within several domains.

Instances of a class are not required to have the same set of properties. It is possible, for example, to have one instance of a class be persistent, controlled, restricted, atomic and replicated while another instance of the same class is volatile, uncontrolled, unrestricted, nonatomic and local. Objects with this later set of properties are called “basic” objects.

2.5 Primitive classes

The Raven language provides special support for two primitive classes. These classes are `String`, and `Int`. Integer objects are unique in that the declaration of a variable of type `Int` allocates space for a capability as well as creating an integer object and initializing the variable to refer to that object:

```
Int i;
```

allocates an object of class `Int` and assigns its capability to `i`. Additionally, integer variables support the usual C arithmetic operators in their standard pre-, post-, or infix forms.

The specialized support for class `String` is not as extensive (String is much more like an programmer defined class). All that is provided is an intuitive handling of string constants. For example, in:

```
String myString;
myString = ‘‘hello’’;
```

the string constant creates an object of type string and initializes it with the value “hello”.

Generally, a Raven programmer is provided with a set of basic utility classes in addition to these primitive classes. These additional classes do not, however, require any particular compiler support. They are standard Raven classes which have been compiled and and converted into a library simply for efficiency. Different users of the Raven system may replace the provided library with one more suited to their own needs.

2.6 Type checking and invocation

Raven supports static and dynamic type-checking, as well as static and dynamic method binding. In order to support static typing, all variables, parameters and return values must be given a type. This type is the implicit class type which is simply the name of the class or the built-in `Any` type. Either way a variable, return values or parameter may only reference objects that are of the designated type or subtype. Invocations on variables are checked by the compiler to ensure that the invoked method exists, that the parameters are of the correct type, and that the return value is appropriately typed for the context. The Raven compiler uses contravarient type-checking (a conservative type equivalence policy).

Informally, Raven's type checking rules are defined as follows: A variable of type `S` can reference any object of class `T`, if `T` is a subtype of `S`. (The definition of subtype is given in Section 3.6.) The subtype relationship is not explicity defined as part of the inheritance relationship. The subtype relationship is defined by the above rules. For example, going back to Figure 1, any class that includes a weight behaviour as defined by class `Weighable` is a subtype of `Weighable` even though that class may not be a subclass of `Weighable`.

Static type-checking should not be confused with static method binding. Type information allows the compiler to verify that invocations are made in a manner consistent with their definitions, but it does not allow the compiler to determine the precise code which will be run when a given method is invoked. Since a typed variable may contain a capability for a given class or any of its sub-classes, and since methods may be overridden in sub-classes, type alone is not sufficient to identify at compile time the code associated with a given invocation. Method binding is therefore usually done at run-time in Raven.

If the programmer can guarantee that a given variable will be assigned capabilities for objects from a particular class and will *not* be assigned capabilities for objects of any of its sub-classes, then the compiler can determine which code is associated with a given method invocation on that variable and can generate code to call it directly, bypassing the dynamic method lookup scheme.

In order to indicate that a variable will contain capabilities only for objects of a particular class, a `$` (dollar sign) is prepended to the variable name. Of course, it does not make sense to use static method binding with variables of type other then a class type. To see how static/dynamic type-checking and method binding interact, consider the following:

```
Any x;
Set y, $z;

x = y = z = Set::new();

x::add( item ); // use dynamic typing and dynamic method binding
y::add( item ); // use static typing and dynamic method binding
z::add( item ); // use static typing and static method binding
```

Static method binding may also be used on method parameters and instance variables. Variables declared with static method binding may only be assigned capabilities which reference `basic` objects. This is because all non-basic property support is by-passed (Note:

this restriction is not currently enforced by the compiler).

3 Raven Language

In this section we describe the Raven language. Since Raven is a derivative of C, we assume the reader is familiar with the basic C syntax. The syntax is given in extended BNF. Keywords (terminals) are given as would appear in a Raven program. Non-terminals are given using <non-terminal> notation. Optional syntax is enclosed in square brackets [ ]. Zero or more occurrences are given in braces {}. If a square bracket or brace exists as a terminal, it is enclosed in quotes (i.e. "[", "]", "{", "}").

3.1 Class Specification

As seen in the example in Figure 1, classes are defined using the class construct as follows:

```
[public] class <class name> [<parameters>] [<- <supertype>] [<properties>]
"{" <instance data> <method defn> "}"
```

All class names must be globally unique. The <supertype> clause is a <type> declaration which indicates the class being inherited from. Section 3.6 describes the structure of type declarations. If the super-type definition is omitted, then it is assumed that the class exists directly under the top object class, Object. Currently only single inheritance is supported.

If the keyword public is used before the class definition then all the instance variables are assumed to be public. Public variables can be read or written directly (See below.)

Classes may optionally be type parameterized. The syntax for the type <parameters> is:

```
‘‘[’’ <identifier> , <identifier> ‘‘]’’
```

The identifiers listed within the brackets then become legitimate type names while in the context of defining the current class, that is, until the next class declaration is encountered. See section 3.6 for an explanation of the type checking rules for parameterized classes.

An object is normally created with a default set of basic properties (See Section 3.11). The option <properties> permits the basic set of properties to be overridden with a new set of mandatory properties.[2] These mandatory properties apply when an object is created and may not be overridden via the pnew method. Only properties which are not explicitly specified as part of the class definition may be used in the pnew method. If a non-mandatory property is not specified at object creation time then the object will be created with the default property as specified in Section 3.11.

3.2 Instance Data

The syntax for the instance data is:

```
<type> [$] <identifier> [public] [class];
```

Type specifications, `<type>`, are described in section 3.6. The `$` (dollar sign) implies that static invocation will be used. This is only valid if `<type>` is not a `Any`. In this case it is assumed that `identifier` refers to a basic object. If the keyword `public` is used, then other objects are permitted access to that instance variable through method invocation. For example:

```
Int priority public;
```

defines a public instance variable which could be accessed as:

```
object::priority = HIGH;
value = object::priority;
```

In this notation, the name of the instance variable implicitly defines the name of a method for getting or setting the instance data. No arguments are passed. If the access is on the left-hand-side of an assignment then the instance variable is set. If it is on the right-hand-side the current value of the instance variable is returned.

This notation is also used when programming the methods within the class. Refer to Figure 1 for an example. In this case, it is not necessary to place the keyword `public` after the instance variables since they are only accessed by the member methods. Within the method definitions of a given class, instance variables may be accessed simply by name. Thus, if the class `List` is declared with an instance variable `size` which records the number of elements in the list, then a behaviour which returns the size of the list could be defined as:

[2] Only a list (white space separated) of property keywords is valid in the `class` construct.

```
behaviour Size {
  return size;
}
```

The keyword class indicates that the instance variable in question is part of the instance data of the class object being defined, rather than of the individual objects of that class. Access to class instance variables occurs via instance objects. For example, if mySet refers to an instance of class Set, and if Set were defined with an instance variable mode indicating something about the mode of operation of the class, then the value of mode could be obtained by examining mySet::mode. Despite the fact that class instance variables are accessed through individual instances of a given class, the actual values are stored in the class object itself, and hence are shared amongst all instances of that class. Class variables are read-only except in the context of the class constructor, which is described in section 3.4.

3.3 Methods Declaration

There are two different types of method definitions. They are:

```
behaviour [copy] [<return type>] <name> ( <parm list> )
       [no lock | read lock | write lock] [private] [class];
constructor( <parm list> ) [class];
```

The first definition defines a "normal" method. A behaviour always returns a value. <return type> is a <type> declaration which indicates the return type of the behaviour. If it is omitted, then the type of the return value is assumed to be Any. The return type specification should also be omitted for behaviours which do not return a value, however, the compiler cannot correctly check that the return values of such behaviours are not used.

The behavior definition includes a variety of keyword modifiers. The copy keyword indicates that when a value is returned, rather than simply returning the capability provided on the return statement, the object referred to by that capability should be copied and the capability for this copy should be returned to the invoker.

If private is used, then the method may only be used within the methods for this class.

The nolock and writelock options are described in section 3.13.

The class keyword indicates that the given behaviour is part of the interface for the class object of the class being defined, rather than part of the interface for instances of the class. As with class variables, class methods are accessed via an instance of the given class.

The syntax for `<parm list>` is:

```
[<argument>] {, <argument>}
```

where `<argument>` is defined as:

```
[copy] <type> [$] <identifier>
```

When a capability is passed in a method invocation only the reference is passed. If the method crosses an address domain (within or between machines) then it is possible to pass a copy of the entire object rather than a reference to the object. This is primarily done for reasons of efficiency and is indicated by the `copy` keyword. If `copy` is used used then all objects that are referenced in the instance data of the first level object are also copied. This copying operation is applied recursively. Copying does not imply migration or replication. What is implied by copying is that an entirely new object(s) is created inside the destination address space.

The `$` (dollar sign) has the same meaning as before. That is, invocation on `identifier` will be be static bound.

3.4 Object Creation / Destruction

The second form of behaviour declaration provides the interface for two initializer methods which can be associated with each class. The `constructor` method is called when an object is first created and is typically used to initialize the instance variables. The optional `class` keyword indicates that the constructor is for the class object being defined rather than instances of that class. Currently, in the Raven system, all classes are created at startup time so it is only during this initialization phase that a class constructor will be called.

Being called during the initilization phase places certain restrictions on class constructors. A class constructor can invoke pre-defined classes (those given in the Appendix), however if it invokes another user-defined class then that class can not invoke it back either directly, indirectly or implicity. The direct and indirect case means no recursive calls that start at a class constructor. The implicit case is when a class A's constructor invokes a method in class B and class B's constructor invokes a method in class A. Even though this case does not imply recursion, it does imply that both class A and B must exist before they can be used. However, in this case, the creation of both depends on each other.

Objects are created by using the method `new`. Objects are removed via garbage collection.

```
<var> = <class name>::new([<constructor args>])
<var> = <class name>::pnew(<properties> [, <constructor args>])
```

The `<class name>` is the capability for the class. `<constructor args>` are optional. The `<constructor args>` must match one for one to the formal arguments defined for the constructor method in the class definition.

There are two variations of the new method. The first, `new`, creates an object of class `<class name>` as defined by the class definition. The `pnew` method creates a non-basic object by overriding the default properties.

The `<properties>` argument in `pnew` is one or more of the properties defined in Section 3.11 separated by the `&` symbol. The keyword `basic` may also be given.

3.5 Method Invocation

We have already seen several examples of method invocation. The complete syntax for method invocation is:

```
[forward] <cap> ::  <method> ( [<method args>] )
```

where `<cap>` is an expression which evaluates to the capability for an object. It is this object upon which invocation is being performed. `<method>` is an identifier which names the behaviour being invoked. `<method args>` is an optional comma separated list of parameter expressions which will be provided to the invoked behaviour. The type checking rules for these parameters are described in section 3.6.

Raven also provides a mechanism for dynamic method invocation:

```
[forward] <cap> ::* <var> ( [<method args>] )
```

This is similar to above except that `<var>` must be the name of a variable of type `String`. The *value* of the variable determines which method is invoked. Because the compiler cannot determine which method is being invoked with this style of invocation there is no type checking of either the parameters or the return value.

Any method invocation can have the keyword forward preceding it. If this keyword is used then the method will not return to the statement immediately following the invocation. That is, the call activation stack will be detached at this point; i.e. a method call without a return. When the invoked method issues a return statement, control is returned to the method or procedure that is "one higher" in the call activation stack. If that method was also called with forward it will be "two higher" and so on. forward is particularly useful in implementing up-calls where there are many method invocations and each invocation is the last statement in the invoking method (and no return value is expected). If the invoked object is controlled, then the detach is not performed until the lock is obtained. This implies that if the invoking object is controlled, the invoked object's lock is obtained *before* the invoking object's lock is released.

3.6 Type Checking

In Raven, a <type> specification has the following form:

```
<type name> [ ``['' <type> , <type> ``]'' ]
```

where <type name> is an identifier which is either Int, Any, or the name of Raven class which has already been declared. Forward references are not permitted in Raven. The optional bracket-enclosed type list contains the type parameters assuming that <type name> is a parameterized class. Suppose that Car has been declared as a non-parameterized class and that List has been declared as a class taking a single parameter, then the following would be legal type specifiers:

```
Int
Car
List[Car]
List[List[Car]]
```

A non-parameterized type is called a *simple* type. The type rules for simple types are developed first and the rules for parameterized types presented as en extension of these.

Every value and every variable in Raven has a type. The types of variables are specified when those variables are declared, and the types of values are computed by the compiler based on the types of the components of the value. If the compiler has sufficient type information, then it will ensure that variables and values are used in a type-consistent manner. The type rules used by the Raven compiler create a contravariant type system.

As implied in the previous paragraph it is not always possible for the compiler to perform type checking. The circumstances under which type checking is not possible are well-defined. The built-in type `Any` is effectively a type which indicates "unknown type". If an expression has type `Any` it indicates that virtually all type checking is disabled. In fact, the only rule which the compiler enforces with regard to this type is that an expression of type `Any` may not be assigned to an expression of type `Int` (for the purposes of this and all further type checking rules, the passing of parameters on method invocations is treated as though an assignment of each actual to its corresponding formal parameter were being peformed).

As noted in section 2.5, the type `Int` also receives some special treatment. No class may inherit from `Int`. The standard C arithmetic operators are provided in their normal form and may be used only with integers. This includes all of the C assignment operators. No support is provided for overloading these operators for other classes. Certain features of the language also rely upon built-in support for integers. The expressions used in the `if`, `while`, `do`, `for switch`, and `case` must all be integers. Integers may be assigned to variables of type `Any`, but the built-in operators may not be used on these variables.

Any simple type other than `Int` or `Any` is a reference to a class defined within Raven. The rules for such types are derived from their class declarations. A class declaration for class `<name>` defines which methods may be invoked on an expression of type `<name>`, the type of the value returned by that invocation and the number and type of parameters that must be supplied on that invocation. Since there are no built-in operators in Raven, aside from assignment and the arithmetic operators for integers, the type system can be fully described simply by addressing these issues.

Informally, Raven's type checking rules are defined as follows: A variable of type `T` can reference any object of class `S`, if `S` is a subtype of `T`. `S` is a subtype of `T` if `S` is identical to `T` or:

1. `S` provides at least the behaviours of `T`.
2. For each behaviour in `S`, the corresponding behaviour in `T` has the same number of arguments and results.
3. The type of the result of `S`'s behavior is a subtype of the type of the result in `T`'s behavior.
4. The types of the arguments of `T`'s behaviors are subtypes of the arguments in `S`'s behavior.

5. S contains at least the public instance variables as T.

6. For each public instance variable in S, the corresponding public instance variable in T has the identical type.

Note that the subtype relationship is not explicity defined as part of the inheritance relationship. An exception to these rules is: If the variable is declared as having static method binding ($) and type `type1`, then an expression of `type2` may be assigned to it iff `type1` and `type2` are idential (i.e., the same class)

The rules for checking of returned values and checking of parameters can both easily be reduced to rules for assignment. Returned values are checked as though performing an assignment of the value to be returned to a variable of the declared return type. Parameters are checked as though performing an assignment of each actual parameter to its corresponding formal parameter. Of course, the number of parameters provided must agree with the number declared.

When an invocation of method `name` is performed on an expression of type `object_type`, the compiler must verify that `name` is a valid method for `object_type` and must retrieve the specified return type and the formal parameter list in order to perform further type-checking. In order for `name` to be a valid method for `object_type` there must be a declaration for behaviour `name` in the class referred to by `object_type`, or in one of the classes from which it is descended. Behaviour `name` may, in fact, be declared in more than one class in the chain of class definitions leading from `Object` to the class referred to by `object_type`. Rules defining when such overriding of behaviours is legal are described below. For the purposes of type-checking an invocation the behaviour definition used is the first one encountered when following a path upwards from `object_type` through the inheritance tree.

When `class2` is declared as inheriting from `class1`, then `class2` may contain declarations for behaviours which are declared in `class1` or in any of its ancestors in the inheritance tree. However there are certain restrictions when overriding a behaviour in this fashion. Suppose that the behaviour being overridden is `name`. Furthermore, let `orig_return` be the return type for the first behaviour declared as `name`, found when traversing the inheritance tree upwards from `class1`. Finally, let `org_p1, org_p2,...` be the types of the parameters of that behaviour. A behaviour override is legal iff the return type for `name` as declared in `class2` is a sub-type of `orig_return` and for the types of each of the parameters `p1, p2...` as declared in `class2`, `p1` is a super-type of `org_p1`, `p2` is a super-type of `org_p2`, etc. In this context, super/sub-type are true of equal types. Of course, it is assumed that the number of parameters is the same in the orginal and the new declaration.

Instance variables are also inherited in Raven and can be similarly overridden. As with overriding behaviours there are restrictions on the type that can be assigned to the variable being overridden. The new type must be identical to the overridden type.

Like C, Raven allows the programmer to override the type information computed by the compiler with a cast operation. The syntax for casts is:

```
( <type> ) <expr>
```

This informs the compiler that `<expr>` is to be treated as though it had type `<type>`.

3.7 Parameterized Classes

As stated elsewhere, Raven permits the definition of parameterized classes. The most intuitive approach to parameterized classes is to consider a single parameterized class as a template for an entire set of related classes. For example, consider the external interface for the classic stack object:

```
class Stack[X] {
  behaviour Push(X item);
  behaviour X Pop();
}
```

As noted in section 3.1, `[X]` appended to the class name indicates that this is a parameterized class. Note that within the class definition the type parameter `X` can be used as though it were a valid type name. Now consider the type declaration which refers to this class:

```
Stack[String] myVar;
```

Note that the type parameter has been replaced with valid type name. This declaration establishes a binding between the type parameter `X` and the type `String`. In performing type checking involving the variable `myVar`, the compiler essentially creates a "virtual" class declaration by taking the original declaration for `Stack[X]` and replacing all occurences of `X` with `String`. Type checking then proceeds according to the rules given in section 3.6.

Within the context of a parameterized class definition, variables or expressions whose type is a type parameter (e.g. a variable within `Stack[X]` having type `X`) are treated for purposes of invocation as if they had type `Any`. This allows the programmer to invoke any method on such a variable as type-checking is effectively disabled. The reason for this is that Raven does not allow constrained type parameters, in which the programmer specifies a minimum set of requirements on the type(s) used to parameterize a given class.

The only additional type-checking rules required by the introduction of parameterized types are those which govern when one class is a sub-class of another.

If `X` and `Y` are type parameters, that is, if in the context of the current class definition they appear in the list of comma separated identifiers following the class name, then `X` is a sub-class of `Y` iff `X` and `Y` are lexically equivalent (i.e. the same identifier). Intuitively, the reason for this is that `X` and `Y` may be bound to arbitrary types, hence, it is impossible to establish any relationship between them unless they are guaranteed to be identical. A type parameter is never a sub-class of a non-type parameter and vice-versa.

If `C1[X1, X2...]` and `C2[Y1, Y2,...]` are both parameterized types, then `C1[X1, X2,...]` is a sub-class of `C2[Y1, Y2...]` iff the virtual class corresponding to `C1[X1, X2...]` is a sub-class, using the rules of section 3.6, of the virtual class corresponding to `C2[Y1, Y2...]`. For example, consider the following class definitions:

```
class Foo[X] {
  ...
}

class Fee[Y] <- Foo[Y] {
  ...
}
```

Parameterized type `Fee[String]` corresponds to virtual class `Fee_String` which inherits from virtual class `Foo_String`. Thus, `Fee[String]` is a sub-class of `Foo[String]`. Note that the compiler never actually generates a class with the name Fee_String. That name is used simply as an aid to understanding the type checking rules which the compiler enforces.

The current Raven compiler has a subtle limitation with respect to parameterized types. Intuitively, the manner in which objects of a parameterized class should be created would be like:

```
newList = List[String]::new();
```

The Raven compiler cannot accept this form of invocation, however. Instead, the following form is used:

```
newList = List::new();
```

Since the type parameterization information is lacking on this invocation, the compiler cannot perform type checking for the constructor parameters.

3.8 Choices

Raven supports one other form of type definition. This definiton permits a variable to refer to a choice of types. The general syntax is:

```
choice <name> "{" <a comma separated list of type name> "}"
```

A variable defined as a choice can refer to any object whose class type is type compatable with one of the choices. The `choice` statement is similar to the `Any` type, however you can use `choice` to restrict the number of possible classes a variable can refer to.

For example,

```
choice Result { String, Int }

Result r;
```

In this example, `r` can refer to any object of class `String`, or of class `Int`. The behavior `isClass` can be used to determine the class of the object that is actually being refered to.

A variable that is defined as a choice type can assigned to another variable if their types are compatible. The following examples indicate what is or is not valid.

```
Result s,r;
Int i;
String q;

// the following assignments are legal
s = r;
r = s;
r = i;
r = q;

// the following are illegal

i = r;
q = r;
```

The above can be accomplised however by:

```
i = (Int) r;
q = (String) r;
```

3.9 Miscellaneous Language Features

Raven has a small collection of predefined identifiers and other miscellaneous features. The identifier nil evaluates to a capability for the nil object. The identifier me evaluates to a capability for the current thread of execution. The type of me is Thread. The identifier self evaluates to the capability for the object within which the current method is executing. The expression '<class name>' evaluates to the capability for the meta-class of class <class name>. Note that parameterized classes are represented at run time by a single class and meta-class (at least, in the context of a single machine), so that type parameters are not specified in this expression. Finally, an expression of the form super::<method>(<parms>) performs an invocation of behaviour <method> as it is defined by the current object's super-class. This allows access to behaviours that have been overridden in a sub-class.

3.10 Access Controls

An object can be created with or without access controls. If it is created with access controls, then it is possible to create multiple references to the object, each with differing invocation rights. Access control is based on the right to invoke the object's methods. The granularity of control is on a per method bases.

When an object is first created, the capability returned has complete access rights. Via the alias method, alternate capabilities for the object can be created that have restricted rights. The syntax for the alias method is:

```
<alias cap> = <cap>::alias( <rights-list> )
```

where <rights-list> is defined as a list of behavior names that may be invoked using the alias capability. The alias is treated in the same way as the primary capability. An object cannot be garbage collected until all references to the object, including those through aliases are removed.

An example where this kind of access control is useful is in connection-based servers. A mail server, for instance, could advertise access via a public directory. The capability returned from the public directory has only the rights to "sign-on" to the mail server. The "sign-on" method accepts the capability of the client, validates it in its local directory and creates an alias which permits further further access via methods such as post or deliver. The alias capability can be disposed of at any time, terminating access from the client, without removing the mail server itself.

3.11 Object Properties

When an object is created it is given properties based on the properties specified as part of the class definiiton or provided explicitly as an argument to the `pnew` method described in section 3.4. There are five categories of properties. Each of these categories has two or more mutually exclusive properties. Any combination of properties, one from each of the five categories, is acceptable. In the definitions below, the property that is underlined is the property assigned to that object at creation time provided no property for that category was given as part of the class definition or supplied as an argument to `pnew`. A property that was explicitly specified as part of the class definition may not be overridden.

Properties are assigned dynamically at object creation time. It is therefore possible to have objects from the same class that have differing properties. In some cases, the class definition "hard codes" a property. Most classes, however, are defined independently of any particular property.

Category: `storage`

`[` <u>`volatile`</u> `| persistent ]`

An object is either volatile or persistent. If an object is volatile its instance data is kept only in RAM. If the system fails or is restarted volatile objects no longer exist. Most objects will be volatile. A persistent object is one that is kept permanently on disk. As such, a persistent object survives system failures and restarts. An object that is defined as persistent is written to disk automatically whenever a method modifies its instance data. It is not guaranteed, however, that the current state of the object will be on disk at the time of system failure. Persistent objects are only written out periodically by the system. These are the same semantics as the Unix file system. A persistent object's capability also survives system failure or restart.

When a persistent object is written to disk, all its instance data is serialized via the implicit (compiler generated) or explicit (programmer generated) class encoding and decoding routines. If the instance data references other objects that are volatile and exist within the same address space as the persistent object – such objects are called dependent – then these objects are also written to disk along with the persistent object. In fact, the transitive closure of all dependent object originating from the persistent object are written out.

Once a dependent object is written to disk it is marked as inaccessible. That is, method invocation is not permitted on dependent objects that have been moved to disk. When a persistent object is mapped back into memory (as a result of a method invocation), the inaccessible flag of all its dependent objects is removed. This implies that the method call chain must include the persistent object before a dependent object.

Category: `concurrency`

`[ uncontrolled |controlled ]`

An object that is defined as uncontrolled has no system locking associated with it and it would be expected that only a single thread of control would be accessing the object, or that concurrent access to the object without system level locking is acceptable. If a class is explicitly designated as uncontrolled then it is the class implementor's responsibility to ensure that the object performs appropriately when it is accessed concurrently. If instead an object is created as uncontrolled, through the use of `pnew`, and has no concurrency property specified as part of the class definition, then it becomes the object user's responsibility for controlling concurrent access to the object.

A controlled object is one that is expected to have several threads attempting simultaneous access. If an object is created as controlled then the system automatically generates a lock for the instance data. Additionally the methods on the object are defined as read if they only access the instance variables and write if they modify the instance data. When a method is invoked on a controlled object the system will automatically request a read or a write lock depending on how the method accesses the instance data. If the lock is granted then execution continues, otherwise the thread is blocked until such time as the lock can be granted. See Section 3.13 for a more detailed discussion of concurrency.

Category: `security`

`[ unrestricted | restricted ]`

An object that is restricted is subject to access controls. The access controls are based on the right to invoke the methods of the object. If the object is unrestricted, all methods are accessible. See Section 3.10 for more information.

Category: `recovery`

`[` <u>`nonatomic`</u> `|` `atomic` `|` `top-atomic` `]`

An atomic object has the "all-or-nothing" property. That is, only when a method terminates normally are any of the changes to the instance variables kept. If the method terminates abnormally, the state of the instance variables are reset to their state just before the method started. Terminating normally implies that the method executed a return statement (either implicitly or explicitly). A method terminated "abnormally" if it executed the `abort` statement or if the Raven system decided unilaterally to abort the method before its completion.

The syntax for abort is simply,

```
abort;
```

This statement can be used even if the object is defined as nonatomic. In this case, the statement acts as an immediate exit from the method and the instance data is not reset.

Raven uses a simplified variant of nested atomic actions. When a top-level action is started (see below) all invocations on atomic objects are considered part of the top-level action. Therefore modifications to an atomic object are not made permanent until the top-level object completes normally. The locks of all objects are held until the top-level action completes.

If the top-atomic property is used, then a new top-level atomic action is started. A top-level action will also occur if an atomic object is invoked and no top-level action is active. A current action is maintained even though nonatomic objects are involved. When an atomic object invokes a nonatomic object, the nonatomic object temporarily becomes atomic for the duration of the invoke.

Atomic objects normally would also have the properties; persistent and concurrent, however this is not required. For example, a volatile, sequential, atomic object is possible. In this case, the atomic property behaves more like a recovery-block. That is, a method could make many changes to its instance variables and then execute an `abort` statement which would have the effect of resetting the instance variables to the state just before the method was started.

Category: `distribution`

`[ local | migrate | replicate | strong-replicate ]`

Objects that are local cannot be migrated or replicated. If a capability for a local object is passed as an argument on a remote invocation, Raven builds a global reference that can then be used to access the object remotely. If the formal parameters indicate copy then the instance data of the local object is duplicated and transmitted to the destination address space. A complete duplicate object is then created at the destination.

If a method invocation on a migratable remote object occurs, then the object may be moved to the calling site. Migration is an execution time optimization and as a result there is no requirement that a migratable object be moved when an invocation is performed on it. Migration is also subject to access control constraints. If the capability for the object does not have migration rights, the object is not moved and a remote invocation occurs. Access rights are not implemented in the current release of Raven.

If an object is created as replicated then a replica of the object should exist at any site that can invoke a method on the object. The mechanism by which objects are replicated and by which the desired level of consistency of the object instance data is maintained across replicated objects are maintained is still very fuzzy. An object can be replicated using strong or weak consistency.

3.12 Control Statements

Most of the common control statements found in the C language are available in Raven. The syntax of the control statements is the same as the C syntax. Statements which are supported include: `if`, `switch`, `for`, `do`, `while`, `for`, `continue`, and the null statement `;`. The `if` statement is supported with/without the `else` clause. Compound statements are also supported.

The only statement which is not supported is `goto`. The `switch` statement syntax is more restrictive then in C. i.e. control statements can not cross `case` boundaries. Finally, the general `(expr),(expr)` syntax is not supported.

3.13 Concurrency

This section discusses the facilities available in Raven to support concurrent and parallel access to objects.

3.13.1 Instance data locking

In Section 3.11 we saw that concurrent is one of the property categories of an object. The concurrency category has properties, `controlled` and `uncontrolled`. If an object is controlled the system automatically generates a lock to control the access the object's instance data. Then when a method is invoked on the object a lock for the instance data is automatically acquired. The type of lock acquired (read or write) depends on whether or not the method modifies the instance data. When the method returns the lock is released. A lock will also be released when a forwarded method invocation is used. The locking system supports multiple readers or a single writer.

A programmer of a controlled class may override the automatic lock generation for a method by using the keyword `nolock` after the behavior definition in the class construct. In this case, no locking will be done on entering the method. Another keywork `writelock` can be used to force a method to acquire a write lock. The ability to do this is required to help reduce the problems associated with a number of simultaneous invocations on an object which then proceed to upgrade their read lock to a write lock either explicitly or by an invocation of a write method on self. If these execution threads proceed through the object together, a deadlock will occur when they all try to get a write lock. By explicitly allowing a method to acquire a write lock, even though it doesn't need it, it is possible to to fend off some of these types of deadlock scenarios.

When a method is marked as nolock, the correct behavior of the method when concurrent access are made to the object becomes the explicit responsibility of the programmer. The statements `readlock`, `writelock` and `unlock` are provided to permit the programmer to explicilty lock and unlock the object's instance data. The syntax of these three statements is simply:

```
readlock | read lock | writelock | write lock ;
unlock;
```

In all cases these statements only accomplish some action on a controlled object. If the object is uncontrolled they have no effect. The locking statements work in the following manner:

read lock If the object's lock in not held then a read lock is acquired. If the lock is currently held as a write lock it is demoted to a read lock. A read lock will remain as a read lock.

write lock If the object's lock isn't currently held then a write lock is acquired. In the case when the currently held lock is a read lock, this statement results in the promotion of the read lock to a write lock. A write lock remains a write lock. If a read lock is held and a write lock cannot be immediately granted then execution will block waiting for the write lock, but the read lock will not be relinquished.

unlock If a lock has been acquired explicitly by the method, then it can be released with this statement. An implicit unlock of user acquired locks is performed when a method completes.

In all cases when a lock cannot be acquired immediately the thread of execution is suspended until the lock can be granted. A thread of execution does not release any locks it holds when it is suspended waiting for a lock. It should also be noted that each method in a threads execution may hold at most one lock. When read and write locks are performed in a method which already has a lock the exisiting lock is converted to one of the specified type. Within a method locks do not stack or form blocking regions like a blocking constuct in a programming language does.

3.13.2 Mutliple threads of execution

When a program starts, it runs in a thread created by the Raven kernel. It is possible to create multiple threads of execution by using the Threads class. The following example demonstrates its use.

```
Thread thread = Thread::new(priority);   // create a new thread object

thread::start(object,method);         // start thread running in object @ method
thread::kill();                       // kill thread
```

Each thread of execution can refer to itself via the keyword `me`. `me` is of type capability; it is the thread equivalent of self. It is always possible for a thread to suspend its own execution via the statement:

```
me::suspend( lock | unlock );
```

The currently executing thread is suspended and optionally unlocks the object. Another thread can later execute the statement:

```
<cap>::resume();
```

Since `me` can be accessed as a regular capability it can be placed in any user defined data structure. The stored value can can then be accessed and used to identify the thread to resume. Of course, it may be necessary for the lock (either read or write) to be reclaimed before execution can really begin.

3.13.3 Early Result

In C the `return` statement conceptually serves two functions, one it returns execution control to the caller and two, it optionally returns a result. In Raven these two functions can be separated. By separating these two functions it is possible for an object to return a result, thus unblocking the invoking object, while maintaining its thread of control until performing a return. During an early result scenario, execution control is transferred to the invoking object as expected and, after some period of execution, the invoked object returns a result. In returning this result the invoking object is unblocked for execution and, in addition, the originally invoked object continues its execution. This results in both the invoked and invoking object executing in parallel. To issue an early result the following construct is used:

```
result [<cap>];
```

After executing `result` the object continues to execute and keeps any locks that it had. The value of the `me` variable changes in the invoked object after a `result` statement is executed to reflect the fact that a new thread of execution has commenced. To terminate this new thread of execution in the invoked object, the object must either execute a specific return, or an implicit return by running off the the end of the code. Once a `result` statement has been executed, the value associated with any `return` statement is ignored.

3.13.4 Delayed Result

The complement to early result is delayed result. With a delayed result the method releases all its locks and terminates without issuing a response. It is expected that another method within the object will perform the result. To exit a method with the intention of performing a delayed result the object uses the following statement:

```
leave;
```

Leave will cause the method to release any locks it has implicitly acquired and the method to terminate. Execution control, however, is not returned to the initiating object and its execution remains suspended until some other thread performs the result that the invoking object is waiting for. Since there may be multiple threads of control waiting on a a delayed result, a way to identify each of these delayed result threads is required. Each thread has a me variable associated with it, (page 37) which is a capability identifying the thread of execution. To accomplish a delayed result the method must save the value of the me variable before doing the leave so that the suspended thread can be identified. Assuming that the me variable has been appropriately saved the actual delayed result is accomplished in another thread in the following way:

```
result "["<cap> [<behaviour name>] "]" [<cap>];
```

Where <behavior name> optionally identifies the behavior. This is only required if type-checking is desired. The first capability is the one identifying the thread of control to perform the result to and the second capability is the result to be returned.

When result is executed control is delivered to the associated waiting object and its execution thread is resumed. The thread of control responsible for completing the delayed result also continues to execute.

3.13.5 Certificate and Companions

In sequential object-oriented languages, an object making a method invocation on another object must wait for a result before continuing. As pointed out in the previous section one way to introduce parallelism into a sequential object oriented language is to allow for an early result within a method. In this situation an invoked method sends back a result but continues its own processing. This allows the original invoker to proceed in parallel with the invoked object. For some objects it may not be possible to construct an operation that can respond with an early result; consequently, no parallelism is even possible for interactions with this type of object. Additionally, relying upon the remote object to provide the parallelism isn't in keeping with the object-oriented design philosophy of hiding the implementation details of a method from the user of the method. Furthermore, it is a somewhat risky endeavor to trust parallelism to another object since a change in the implementation technique of the methods for that object could eliminate or reduce the amount of parallelism within the system. To combat this problem it is essential that the invoker of an operation have the ability to specify whether or not the invocation of a particular method should be allowed to proceed in parallel with the object's own execution. It is the desire to provide a facility to permit this form of parallelism that supplies the impetus for `certificates` and `companions`.

A companion is a separate thread of execution that runs in parallel with the thread of execution it was derived from. A companion is compromised of a set of method invocations and these methods will all be executed in parallel when the companion is started. The net effect of starting a companion is that all the methods specified in the companion run in parallel with themselves and the thread of control which started the companion.

Once a companion has been created, a `certificate`, and its associated methods, can be used to track the progress of the companion. Each time a companion is created a corresponding certificate is generated and can be obtained by the programmer. A certificate is an object which responds to a number of methods that can be used to monitor the created companion. The syntax for the creation of a companion is:

```
<companion> ::=  !{ <method_invoke_list> }! [<tag>] ;
<method_invoke_list> ::= <method_invocation> |
                         <method_invocation> <method_invocation_list>
<method_invocation> ::= [<cap> = ] <cap>::<method>(<method_args>);
<tag> ::= "["basic type"]"
```

The only type of statements that may appear in the companion are method invocations. The capabilities being invoked upon must appear as simple variable names (i.e. identifiers), as must the variables on the left hand side of the assignment if present. For the purposes of determining the values of the arguments to the method invocations the initiator's execution is suspended until all the parameters for the methods have been evaluated. This insures that the variables used in the evaluation cannot be changed unexpectedly by the initiator. Just as in C, the order of parameter evaluation for a method invocation is undefined and within a companion the evaluation order of the set of parameters of all the method invocations is undefined.

Within a companion it is possible to specify multiple invokes on the same object. When this is done the requests are presented to the object in the order in which they are specified within the companion. This, however, says nothing about the ultimate order of execution of the requests. The object itself may delay or postpone the method execution implicitly because of locking restrictions in force at the method level or explicitly through the delayed result facility.

As indicated previously the creation of a companion has implicitly associated with it the generation of a certificate. The syntax for acquiring a certificate is:

```
<cap> = <companion>
```

In this situation the capability returned is of class Certificate. There is no requirement that the certificate generated by the activation of a companion be assigned to a variable, it may simply be ignored. An example use of the companion and certificates facilities is shown below:

```
     cert = !{ result_1 =  object_1::method(arg1, ... );
                                              .
                                              .
                                              .
              result_n = object_n::method(arg1, arg2, ... ); }! tag_value
```

In this example the variable `cert` is assigned the capability that identifies the companion that is created. This certificate is given an initial tag value. The optional tag field is a recognition that a parallel request needs some way of being tagged for later identification. Although the certificate itself partly serves this purpose the tag extends this by acting as a grouping function. It is easier to make a classification and include it as a tag at the time the certificate is generated then to have to build special data structures to perform the unmapping later.

In addition to an individual certificate, there are objects of class `Certificates` which are simply groups of `certificate`s. A certificate object is more than just the object identifier for the companion: it contains within itself additional data to aid the programmer in using companions and certificates. Each certificate has the following properties:

1. A certificate object is uniquely identified because it is a capability.
2. It can hold a tag value that is reserved for the exclusive use of the programmer.
3. It maintains a status part, which gives some indication of the execution status of the companion.

The usefulness of certificates is a function of the methods that can be invoked on them. The following is an outline of the certificate methods and an indication of whether they apply to a single certificate or a group of them. (See Appendix for details.)

1. `Assignment` - (certificate and certificates) - Assuming proper class compatibility, the right-hand side is assigned to the left-hand side. An object of class certificate may be assigned to a certificate group, but not the reverse.

2. `add/delete` - (certificates) - These methods allow for the dynamic manipulation and updating of objects of class certificates. The addition operation will augment the existing object with all the certificate objects specified which are part of the current object. Deletion operates in the same manner, only it deletes a certificate object.

3. `getStatus` - (certificate) - This method takes a single certificate as its value and returns the current status of the companion.

4. `getTag` - (certificate) - This method takes a certificate and returns the integer tag.

5. `setTag` - (certificate) - This method takes a certificate and sets the tag value to the integer value.

6. `equals` - (certificate and certificates) - The equality operator returns `TRUE` if the certificate identifier part of the right and left hand sides of the equals sign identify the same companion, otherwise `FALSE` is returned.

7. `Iteration` - (certificate and certificates) - The iterator operator is the mechanism used to block and wait for a companion or collection of companions to complete. Classes in Raven which support iteration are expected to adhere to a set of standards designed to provide a uniform way of accomplishing iteration. The technique being used requires an initial method invocation to initialize the iteration and retrieve the first item. Subsequent items of the iteration are retrieved through method invocations to get the next item. The usage of the iterator operator in conjunction with certificates and companions is best illustrated with an example.

   For the example assume that two variables, `g` of class Certificate, and `G` of class Certificates exist and that `G` has some certificate objects in it. (For the sake of completeness `G` may also be an object of class certificate.) To iterate on the certificates the following example of Raven code illustrates the process.

   ```
   for (g = G::firstItem(); g != nil; g = G::nextItem()) {
     execute the body of the for loop;
   }
   ```

   The basic behavior of the iterator is that each item that is a component of the thing being iterated on is retrieved once. Briefly then, the behavior of the iterator on class certificate and certificates is defined as follows:

   - Unless an explicit break is done out of the iterator loop, each certificate in `G` will be selected once and only once.

- `g` will have the value of the current certificate selected by the iterator operator.
- A certificate can be selected only if the companion operation it is associated with has completed. If a companion contains multiple method invocations then all the invocations must be complete for the companion to be viewed as being completed.
- When there are no completed certificate operations, the invocation of the nextItem() method on `G` will block waiting for a companion to complete.
- The iterator will not block waiting for the companion associated with an element of `G` to complete when other completed members of `G` have not been returned by the iterator yet.

Together, the facilities of thread creation, suspend and resume, early and delayed result, combined with certificates and companions provide a powerful set of tools to exploit parallelism when programming in Raven.

3.13.6 Locking Revisited

The facilities for parallelism combined with the automatic locking described in section 3.13.1 produce some interesting problem scenarios. The three problem areas are:

- An object does an invocation on itself.
- An object holding a lock does an invocation on itself and the the invoked method does an early result.
- An object holding a lock invokes on itself and the invoked method does a delayed result.

Recursive method invocations on an object only present a problem when a write lock is already held or is requested. If only read locks are involved no problem exists. To minimize the effects of recursive invocations, each execution thread is assigned a globally unique ID that can be used to track chains of invocations. The main purpose of this ID is to make it possible to determine when a method invocation attempts to acquire a lock on an object that is already locked within the invocation chain. When a recursive invocation involves a write lock, the lock will be granted if the releasing of all the locks held on this object by the invocation chain would permit the lock to be granted.

The problems are more complicated when a recursive invocation involves an early result. Since an early result could result in two threads of execution, unexpected changes to the instance data through these recursive invocations are a possibility. To eliminate this for early result, the result is only returned and a separate thread created if no other locks are held on this object in the invocation chain, or all the locks are read locks. When there are a mixture of read and write locks held on the object by the invocation chain the result of the early result is delayed until the method with the early result in it does its return. This eliminates the parallelism between the invoker and invokee that early result provides, but maintains the proper invocation semantics.

A similar problem exists when a recursive invocation has a delayed result. When the leave statement is executed the method terminates and the instance data lock acquired to enter the method is released. The execution of the invoking method blocks waiting for the leave to be completed. However, locks on this object could be held by other methods in the invocation chain and this could prevent another invocation chain from invoking the method that would result in the completion of the delayed result. To avoid this, when the leave is done all the locks in the invocation chain that are associated with the object are temporarily released. This permits other methods to acquire locks on the object with the expectation that one of them will complete the delayed result. When the delayed result is completed the suspended thread can be restarted provided all the locks it held on the object can be granted. If the locks can't be regranted the "delayed" thread will remain suspended until they can.

4 Building an application

This section provides a brief overview of how a Raven application can be built. The procedure to build an application, as much as possible, tries to follow the steps that would be undertaken if one were writing the application in C.

If a program is to be executed it must have a well known location to begin execution at. In C this is the main() procedure. Raven has this C equivalent in the form of the Main class with the well known behavior start. For every Raven program, the programmer must define a class Main and the method start. When the Raven system starts it automatically creates one instance of the class Main and then invokes the start method on that instance. The start behavior provides the same functionality as the main body of a regular C program.

Further information on how to use the Raven compiler can be found in Appendix C.

5 Syntax

The current syntax for the Raven language is provided below. A form of extended BNF is used to express the grammar. Since the compiler has been constructed using YACC/LEX, a standard LALR(1) grammar can be obtained by contacting the authors. The extended BNF form was thought to be easier to read. Terminal symbols are presented in bold face. Usually these correspond to the keyword with the same name as the terminal symbol, or to the punctuation symbol named. Optional terms are enclosed in brackets []. Terms enclosed in braces {} may be repeated zero or more times. The terminal **ident** indicates an identifier (alphanumeric string beginning with an alphabetic character). The terminal **type** indicates an identifier which has previously been used as the name of a class. Formatting is not significant – it is intended to improve readability.

program : unit

unit : classDecl behaviourDefn

classDecl: [**public**] **class ident** [[**ident** **, ident**]] [< − typeDecl] property
{ instVarDecl behaviourDefn }

typeDecl: **type** | **type** [typeDecl **,** typeDecl] | *** type**

instVarDecl : varDef [**public**] [**class**] **;**

varDef : **type** varName **,** varName

varName : [**$**] **ident**

behaviourDefn : behaviourDecl compoundStmt

behaviourDecl : **behaviour** [**copy**] [typeDecl] **ident (** parm **)** | [**lock**] [**private**] [**class**] |
constructor (parm **)** [**class**]

parm : [**copy**] typeDecl varName

compoundStmt : { varDef stmt }

stmt : **;** | compoundStmt | expr **;** | **return** [expr] **;** |
result [expr [**ident**]] [expr] **;** | **leave ;** |
break ; | **lock ;** | **readlock ;** | **read lock ;** |

writelock ; | **write lock ;** | **continue ;** |
if (expr **)** stmt [**else** stmt] |
while (expr **)** stmt |
do stmt **while (** expr **) ;** |
for ([expr] **;** [expr] **;** [expr] **)** stmt |
switch (expr **) {** case [default] **}**

case : **case** expr **:** stmt

default : **default :** stmt

expr : expr **?** expr **:** expr | expr (**=** | **+=** | **-=** | ***=** | **/=** |
%= | <<= | >>= | ! = | **&=** | **XOR=**) expr | binary

binary : binary (**==** | **!=** | **¡** | **¿** | **¡=** | **¿=** | **!!** | **&&** |
xor | **&** | | | ***** | **/** | **%** | **¿¿** | **¡¡** | **+** | **-**) binary | unary

unary : (| - | **++** | - - | **forward**) unary |
(typeDecl **)** unary | unary **:: ident** [**(** expr **,** expr **)**] |
super :: ident [**(** expr **,** expr **)**] |
unary **:: * ident (** expr **,** expr **)**] |
type :: new (expr **,** expr **)**] |
type :: pnew (properties **,** expr **)**] |
factor

factor : **(** expr **)** | **intconst** | **ident** | **stringconst** | **self** | **nil** |
super | **!{** companion **;** companion **}!** [**[** expr **]**]

companion : [**ident =**] **ident :: ident(** expr **,** expr **)**

properties: property **&** property | **basic**

property : **volatile** | **persistent** | **restricted** | **unrestricted** | **controlled** |
uncontrolled | **atomic** | **nonatomic** | **topatomic** | **local** | **migrate** |
replicate

Appendix A. Class Library

The following class library can be used when building an application. All classes are instances of class `*Class`, except for class `Class` itself which is an instance of class `**Class`, and class `**Class` which is an instance of itself.

```
class Object
  {
  constructor();
  behaviour Int getInstanceDataPtr(*Class classCap);
  behaviour *Class myClass();
  behaviour String myClassName();
  behaviour Int isMemberOf(*Class classCap);
  behaviour doesNotUnderstand(Int methodID);
  behaviour isNotImpl(String message);
  }

class Class
  {
  behaviour Class createNewClass(Class parent, String Name, Int instanceSize,
  Int properties) class;
  behaviour cap new() class;
  behaviour Int canUnderstand(String meth) class;
  behaviour cap pnew() class;
  behaviour String name() class;
  }

class ByteArray
  {
  constructor(Int ssize);
  behavior Int getByte(Int location);
  behavior Int putByte(Int location, Int value);
  behavior Int get4Bytes(Int location);
  behavior Int put4Bytes(Int location, Int value);
  behavior Int putNBytes(cap from, Int start, Int count);
  behavior Int getNBytes(Int start, cap to, Int count);
  behavior Int size();
  behavior Int basicAddress();
  behavior cap basicAddressOfElement(Int index);
  }
```

```
class String <- ByteArray
  {
  constructor (Int size);
  behavior print();
  behavior printStderr();
  behavior Int hash();
  behavior String strcat(String s2);  // Concatenate s2 to this string
  behavior String strncat(String s2, Int count);
  behavior Int strcmp(String s2);
  behavior Int strncmp(String s2, Int count);
  behavior Int  strcasecmp(String s2);
  behavior Int  strncasecmp(String s2, Int count);
  behavior String strcpy(String s2);
  behavior String strncpy(String s2, Int count);
  behavior Int strlen();
  }

class List <- Object uncontrolled
  {
  behavior add(cap item)  nolock;
  behavior push(cap item)  nolock;
  behavior append(cap item)  nolock;
  behavior head()  nolock;
  behavior tail()  nolock;
  behavior Int size()  nolock;
  behavior cap deleteFirst() nolock;
  behavior Int deepDeleteFirst() nolock;
  constructor();
  behavior cap nextItem() nolock;
  behavior firstItem()  nolock;
  behavior cap deleteItem(cap item) nolock;
  behavior cap findItem(cap item) nolock;
  }

class Thread uncontrolled
  {
  constructor(Int pprio);
  behaviour starter();
  behaviour Int start(cap obj, String mmethod, cap aarg1, cap aarg2,
  cap aarg3, cap aarg4);
  behaviour Int attach(cap obj);
```

```
  behaviour Int getPid();
  behaviour Int threadRunnable();
  behaviour Int kill();
  behaviour String threadName();
  behaviour Int resume();
  behaviour Int stackSize();
  behaviour suspend();
  behavior  sleep(Int duration);
  }

#define   IDLE  0
#define   RUNNING 1
#define   FINISHED  2

class Certificate <- Object controlled
  {
  behavior addThread(cap thread);
  behavior threadFinished();
  behavior wait() nolock;
  constructor();
  behavior Int getTag();
  behavior Int setTag(Int value);
  behavior Int equals(cap other_cap);
  behavior Int setNotify(cap cert);
  }

public class CertificateThread <- Thread uncontrolled
  {
  cap cert_to_notify public;
  cap return_value_addr public;
  constructor () ;
  behavior certStart(cap obj, String mmethod, cap return_addr, cap certificate,
  cap parm1, cap parm2) nolock;
  behavior certStarter();
  }
```

Appendix C. Raven Compiler Manual Page

NAME

rvc – Raven language compiler

SYNOPSIS

rvc [options] [file ...]

DESCRIPTION

rvc is the compiler for the Raven language. It accepts Raven source code and converts it to C. Upon conversion, the resulting C code is compiled and linked for execution.

rvc accepts numerious options and file formats to provide flexibiliy for both compilation and linking. Any errors or warnings from *rvc* are sent to stderr. The Raven compiler expects files specified on the command line to have one of the following suffixes:

.r The suffix for a Raven source file.

.c A C file.

.o An object file.

OPTIONS

–v Print out the compiler version number and exit.

–o*file* Place output in *file*. This option does not work in conjunction with the **–r** option. The default output file name is rv.out.

–r Compile to C only. The resulting C code is passed through an C beautification program in an attempt to make the output more readable (mostly a lost cause). For each file with a **.r** suffix, the compiler will produce a corresponding file with the suffix **.c**.

−n Do not generate #line directives. The compiler will normally generate preprocessor directives which indicate which line of the Raven source corresponds to a given location in the resulting C code.

−g Pass the **−g** option (produce symbolic debugging information) along to the C compiler. This implies the **-k** option.

−k Do not delete the generated C files. For each file given on the command line with the suffix **.r** the compiler will produce a corresponding file with the suffic **.c**. This C code is *not* passed through the C beautifier.

−c Do not link the object files to produce an executable. For each file on the command line with suffix **.r** or **.c**, the compiler will produce a corresponding object file with suffix **.o**.

EXAMPLES

rvc a.r b.r c.c d.o

Convert a.r and b.r to their .c files respectively, compile the resulting files along with c.c and finally link all the files with the d.o file, producing an executable in file rv.out.

rvc -v

Print information about which version of the Raven compiler is being used.

BUGS

Not all the features of the Raven language are currently supported, while other features are supported in a very strict way.

References

[Cox86] Brad Cox. *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley, Reading, Mass., 1986.

[EFM+91] Steve Engelstad, Keith Falk, Warren Montgomerry, Joe Neumann, Ralph Straubs, Jim Vanendorpe, and Mike Wilde. A Dynamic C-Based Object-Oriented System for Unix. *IEEE Software*, 8(3):73–85, May 1991.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Langauge and its Implementation.* Addison-Wesley, Reading, Mass., 1983.

[Str86] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Mass., 1986.