Scala and AsmL side by side

Stéphane Micheloud

Programming Methods Laboratory
Swiss Federal Institute of Technology Lausanne
INR Ecublens, 1015 Lausanne, Switzerland
stephane.micheloud@epfl.ch

May 29, 2003
Revisited, August 19, 2005*

Abstract

Abstract state machines (ASMs) describe the dynamic behavior of complex systems in an intuitive and mathematically precise way. AsmL is an advanced ASM-based modeling and programming language. It provides a modern specification environment that is object-oriented and component-based.

We present a brief comparison of the features of AsmL and their counterpart in Scala, a general purpose programming language which combines object-oriented, functional and concurrent elements.

1 Introduction

In this paper we intend to determine to which extent the general-purpose language Scala may be used instead of the domain-specific language AsmL to implement executable ASM-based models.

1.1 AsmL

AsmL is a specification language based on Gurevich's abstract state machines (ASM) [1, 5]. Specifications written in AsmL can be run as programs and are also called *executable specifications*. In contrast to the sequential execution model used by most programming languages AsmL favors parallelism by default.

AsmL can be used to faithfully capture the abstract structure and step-wise behaviour of any discrete system, including very complex ones such as integrated circuits, software components and devices that combine both hardware and software.

AsmL includes a state-of-the-art type system with extensive support for type parameterization and type inference. Using clear semantics, it provides a unified view of classes used for object-oriented programming, in addition

*Updated to reflect changes in the language specification of Scala and/or AsmL to structured data types. It supports mathematical set operations - such as comprehension and quantification - that are useful for writing high-level specifications.

Unlike most popular programming languages AsmL is non-deterministic. Non-deterministic systems exhibit two characteristics: there is a finite set of possibilities (i.e. states) and the result may be any value within that set.

Apart from AsmL source files written either in plain text or in Xml format the AsmL compiler also accepts C# source files so that an AsmL program may consist of AsmL or C# code. But note that nearly everything we can do in C# can also be done directly in AsmL, accessing the .Net framework libraries.

AsmL is being developed at Microsoft Research (MSR) in Redmond by the Foundations of Software Engineering (FSE) group under the supervision of Dr. Yuri Gurevich. The language is bootstrapped, i.e. compiler and other tools are written in AsmL itself. The AsmL compiler is available on the .Net platform[1] which is based on the Microsoft's Common Language Runtime (Clr) and generates a standalone executable (`.exe`).

1.2 Scala

Scala is both an object-oriented and functional language. It is a pure object-oriented language in the sense that every value is an object. Types and behavior of objects are described by classes. Classes can be composed using mixin composition. Scala is designed to interact well with mainstream object-oriented languages, in particular Java and C#.

Scala is also a functional language in the sense that every function is a value. Nesting of function definitions and higher-order functions are naturally supported. Scala also supports a general notion of pattern matching which can model together with case classes the algebraic types used in many functional languages. Furthermore, this notion of pattern matching naturally extends to the processing of Xml data.

Type parametrization, type inference and type covariance are some of the advanced features supported by the Scala type system.

Scala is being developed at the Programming Methods Laboratory (LAMP) of the Swiss Institute of Technology Lausanne (EPFL) under the supervision of Prof. Martin Odersky [7]. The current Scala compiler is written partly in Scala itself and partly in PiCo [10], a Java extension supporting algebraic types. The Scala compiler is available on any Java platform[2] (actually version 1.4 or newer) and generates Java bytecode.

[1] The AsmL software is available from `http://research.microsoft.com/fse/asml/`

[2] The Scala software is available from `http://scala.epfl.ch/`

2 Getting Started

For the purpose of this comparison we will use version 2 of AsmL [3] and the development version of Scala [7] for coding the examples[1] presented in this paper. Most of them have been adapted from [3] and [8].

2.1 Hello World

Let us consider the well-known Hello World program as our first example.

In AsmL the rule `Main()` constitutes the entry point of any AsmL Windows console application.

```
Main()
    WriteLine("Hello World!")
```

As Haskell and a few other languages AsmL uses indentations to denote block structure, and blocks can be nested (AsmL does not recognize tabs, so don't use the tab character for indentations). Multiple instructions inside a block are separated by carriage returns. In our example `WriteLine` is an external operation which simply prints a string without changing any state. By convention of Microsoft's Clr the names of all types and methods provided by the AsmL 2 library are capitalized.

In Scala the special method `main(args: Array[String]): Unit` constitutes the entry point of any Scala program. The `args` parameter gives access to the program arguments as in Java or C#.

```
object Hello {
  def main(args: Array[String]): Unit = {
    System.out.println("Hello World")
  }
}
```

As C-like languages Scala uses curly braces ("{" and "}") to indicate blocks. In Scala braces may be left out if the block contains only one expression. Multiple expressions inside a block are separated by semicolons. Note that we use the Java static function `println(String s)` to print out a string on the standard output; indeed a Scala program can access any Java library.

2.2 Sorting

We now consider a very simple specification of a one-swap-at-a-time sorting algorithm. We don't use global variables as in the original AsmL example [8] as they are *not allowed* in Scala (idem for methods). Again we present here the two programs in AsmL and Scala.

```
Swap(s as Seq of Integer) as Seq of Integer
```

[1] The code samples presented in this paper were all compiled using version 2.2 of Microsoft™ AsmL on Microsoft Windows XP and version 1.4.0.0 of Scala on Fedora Core 3; they can be obtained from the author as a separate archive file (including makefiles).

```
    var A = s
    step choose i in Indices(A),
                j in Indices(A) where i < j and A(i) > A(j)
        A(j) := A(i)
        A(i) := A(j)
    step return A

Sort(s as Seq of Integer) as Seq of Integer
    var A = s
    step until fixpoint A := Swap(A)
    step return A

Main()
    WriteLine(Sort([1,4,7,3,5]))
```

The keyword `step` marks the transition for one state to another. The notion of *state* plays a central role in AsmL and we will take a closer look on it in section 6. The `:=` operator in AsmL is used to update variables; all updates given within a single step occur simultaneously at the end of the step. The `choose` operator picks a pair of elements from a list in a non-deterministic way. `step until fixpoint` precedes a block of statements that will be repeatedly run until no changes result. A fixpoint occurs when two consecutive states are equal.

The same result can be achieved in Scala using functions. By the way this example illustrates the fact that literally rewriting code is generally not the best choice for providing a good solution in a different language.

```
object Sort {

  import AsmL._;

  def sort(xs: List[Int]) = {
    def swap(xs: List[Int]) = {
      val ys = for (
          val i <- xs.indices;
          val j <- xs.indices;
          (i < j) && xs(i) > xs(j))
        yield Pair(i, j);
        if (ys.isEmpty)
          xs
        else {
          val Pair(inx1, inx2) = choose(ys); // ys.head;
          swaplist(xs, inx1, inx2)
        }
    }
    fixpoint(xs, swap)
  }

  def main(args: Array[String]) =
    Console.println(sort(List(1, 4, 7, 3, 5)));
```

```
}
```

The `for/yield` construct in Scala takes a list of generators and filters and returns a list of elements satisfying the specified constraints.

For convenience the functions `indices`, `choose`, `swaplist` and `fixpoint` are defined in a separate Scala module named `AsmL` and are imported using a Java-like `import` directive. `choose(ys)` simply picks an arbitrary pair of elements from the list `ys` and the same result can be achieved in a deterministic way writing `ys.head`.

Unlike in AsmL sequences in Scala are immutable so we need the function `swaplist` to swap two elements of a list.

For the sake of completeness we present here one possible implementation of `AsmL`:

```
object AsmL {

  import scala.collection.mutable._;

  private val rand = new java.util.Random(System.currentTimeMillis());

  def indices[A](xs: Array[A]) = List.range(0, xs.length);

  def fixpoint[A](x: A, f: A => A, eq: (A, A) => Boolean) = {
    def iterate(oldX: A, newX: A): A =
      if (eq(oldX, newX)) newX else iterate(newX, f(newX));
    iterate(x, f(x))
  }

  def fixpoint[A](x: A, f: A => A): A = fixpoint(x, f, (x: A, y: A) => x == y);

  def fixpoint(x: Array[Int], f: Array[Int] => Array[Int]) = {
    def iterate(x: Array[Int]): Array[Int] = {
      var newX = new Array[Int](x.length);
      for (val i <- indices(newX)) { newX(i) = x(i) };
      f(newX);
      if (java.util.Arrays.equals(x, newX)) newX else iterate(newX);
    }
    iterate(x)
  }

  def min[A](less: (A, A) => Boolean, xs: List[A]): Option[A] = xs match {
    case List()  => None
    case x :: Nil => Some(x)
    case y :: ys => min(less, ys) match {
      case Some(m) => if (less(y, m)) Some(y) else Some(m)
    }
  }

  def choose[a](xs: List[a]): a = xs(rand.nextInt(xs.length));
```

```
  def swaplist[a](xs: List[a], i: Int, j: Int): List[a] = {
    def aux(xs1: List[a], k: int): List[a] = xs1 match {
      case Nil => Nil
      case y :: ys =>
        (if (k == i) xs(j)
         else if (k == j) xs(i)
         else y) :: aux(ys, k + 1)
    }
    if (i == j || i < 0 || i >= xs.length || j < 0 || j >= xs.length)
      xs
    else
      aux(xs, 0)
  }

  trait Ensure[A] {
    def ensure(postcondition: A => Boolean): A
  }

  def require[A](precondition: => Boolean)(command: => A): Ensure[A] =
    if (precondition)
      new Ensure[A] {
        def ensure(postcondition: A => Boolean): A = {
          val result = command;
          if (postcondition(result)) result
          else error("Assertion error")
        }
      }
    else
      error("Assertion error");

  def Map[A, B](elems: Pair[A, B]*): Map[A, B] = {
    val map = new HashMap[A, B]; map ++= elems; map
  }

  def Map[A, B](elems: Iterable[Pair[A, B]]): Map[A, B] = {
    val map = new HashMap[A, B]; map ++= elems; map
  }

  def Set[A](elems: A*): Set[A] = {
    val set = new HashSet[A]; set ++= elems; set
  }

  def Set[A](elems: Iterable[A]): Set[A] = {
    val set = new HashSet[A]; set ++= elems; set
  }

}
```

Inorder to initialize the random seed we simply call the static function `currentTimeMillis()` of the standard JAVA library.

3 Predefined Datatypes

The *primitive* built-in datatypes both in Scala and AsmL are numbers, characters and truth values.

The main built-in *composite* datatypes in AsmL are strings, enumerations, sets, sequences, maps and tuples. Unlike AsmL *composite* datatypes in Scala are not built in the language, but are parts of Scala (or Java) libraries or are user-defined.

For a detailed list of predefined types see tables 3 and 4 in the appendix.

Here are some examples of declarations:

```
class Frame

Main()
    weekDays as Set of String = {"Mon", "Tue", "Wed", "Thu", "Fri"}
    directory as Map of String to Integer =
        {"emergency" -> 911, "info" -> 411}
    interval = [1..9]
    frame1 = new Frame
    frame2 = new Frame
    frame3 = new Frame
    stack as Seq of Frame = [frame1, frame2, frame3]
    nameAndAge as (String, Integer) = ("Pythagoras", 2582)

    step WriteLine(interval(1))
    step WriteLine(directory("info"))
    step WriteLine(directory("hotline"))
```

Listing 1: Composites.asml

Similar to arrays in most languages sequences in AsmL and Scala are zero-based, so the element at index `1` has the value `2`. The last line code generates a runtime exception as shown below:

```
2
411

Unhandled Exception: Microsoft.AsmL.IndexOutOfBoundsException: hotline
   at Microsoft.AsmL.Map.get_item(Object d)
   at Application._GLOBAL_.Main() in ...\Composites.asml:line 17
```

We will present the handling of runtime exceptions in section 8.

```
object Composites {

  import scala.collection.mutable._;

  class Frame;

  def main(args: Array[String]) = {
    val weekDays : Set[String] = AsmL.Set("Mon", "Tue", "Wed", "Thu", "Fri");
    val directory: Map[String, Int] =
```

```
        AsmL.Map(Pair("emergency", 911), Pair("info", 411));

      val interval = List.range(1, 10);
        val frame1, frame2, frame3 = new Frame;
        val stack: Seq[Frame] = List(frame1, frame2, frame3);
        val nameAndAge: Pair[String, Int] = Pair("Pythagoras", 2582);

        Console.println(interval(1));
        Console.println(directory("info"));
        Console.println(directory("hotline"))
    }

}
```

Listing 2: Composites.scala

The example in SCALA behaves the same way:

```
2
411
Exception in thread "main" java.lang.RuntimeException: key not found
        at scala.Predef$.error(sources/scala/Predef.scala:32)
        at scala.collection.mutable.Map$class.apply(.../mutable/Map.scala:37)
        at Composites$.main(src/Composites.scala:31)
        at Composites.main(src/Composites.scala)
```

4 Function Definitions

ASML makes a distinction between functions and procedures. Conceptually, functions make no updates to global or instance variables while procedures may or may not perform updates. Functions must return a value; procedures may return a value. The keywords `function` and `procedure` are optional, in which case a function is assumed.

Here is an example both in ASML and in SCALA.

```
Square(x as Integer) as Integer return x * x

Max(i as Integer, j as Integer) as Integer
    return (if i > j then i else j)

Max(s as Seq of Integer) as Integer or Null
    if s = [] then return null
    else return (any m | m in s where not (exists n in s where n > m))

Main()
    step WriteLine(Square(5))
    step WriteLine(Max(2, 3))
    step WriteLine(Max([1, 8, 2, 12, 13, 6]))
```

All method arguments and return values are passed by value.

```
object Funcs {

  def square(x: Int): Int = x * x;

  def max(i: Int, j: Int): Int = if (i > j) i else j;

  def max(s: List[Int]): Option[Int] =
    if (s.isEmpty) None
    else Some((for (val m <- s; ! (s exists (n => n > m))) yield m).head);

  def main(args: Array[String]): Unit = {
    Console.println(square(5));
    Console.println(max(2, 3));
    Console.println(max(List(1, 8, 2, 12, 13, 6)))
  }

}
```

SCALA supports local type inference so the return type of the functions `square`, `max` and `main` can be left out here. SCALA also supports lazy evaluation of function arguments and anonymous functions.

Functions in ASML may contain both pre- and post-conditions; the `require` and `ensure` statements document constraints placed upon the model. Violating a constraint at runtime is called an *assertion failure* and indicates a modeling error. In a post-condition the name `result` refers to the value of the block which contains the pre- and post-conditions.

```
function Arb(s as Seq of Integer) as Integer
    require s <> []
    ensure result in s
    return Head(s)

procedure Main()
    step
        s = [1, 2, 3]
        WriteLine("s = " + s + ", Arb(s) = " + Arb(s))
    step
        e as Seq of Integer = []
        WriteLine("e = " + e + ", Arb(e) = " + Arb(e))
```

The first call to `WriteLine` prints out `s = {1, 2, 3}, arb(s) = 1` and the second throws an `AssertionFailedException` exception.

In SCALA it is the responsability of the programmer to ensure that both conditions are checked at the right place.

A first possibility would be to mimic that ASML construct using two polymorphic functions `require` and `ensure`. A more straight-forward way for checking pre- and postconditions is to use an `assert` function as in JAVA.

```
object Assertions {

  import AsmL._;
```

```
  def arb[a](s: List[a]) =
    require (! s.isEmpty) {
      s.head
    } ensure (result => s contains result);

  def assert(cond: Boolean) =
    if (! cond) error("Assertion error");

  def arb1[a](s: List[a]) = {
    assert(! s.isEmpty);
    val result = s.head;
    assert(s contains result);
    result
  }

  def main(args: Array[String]) = {
    val s = List(1, 2, 3);
    Console.println(arb(s));
    Console.println(arb1(s));

    val e: List[Int] = List();
    Console.println(arb(e));
    Console.println(arb1(e))
  }

}
```

5 Advanced Logic Operators

Scala and AsmL have both existential and universal quantifiers for collection types such as sets, sequences or maps. The variable range can be restricted by means of constraints.

```
Main()
    N = 100
    s = {1..N}

    step WriteLine(exists x in s where x * x = 16)
    step WriteLine(forall y in s holds y >= 6)
```

The output result is `true` respectively `false`.

```
object Logics {
  def main(args: Array[String]) = {
    val N = 100;
    val s = List.range(1, N+1);

    Console.println(s exists (x => x * x == 16));
    Console.println(s forall (y => y >= 6))
  }
```

```
}
```

Set comprehension generalizes naturally to sequences and maps. Comprehension means that you can state all the properties that characterize the elements of a set rather than list them explicitly.

Here is the quicksort algorithm both in AsmL and in Scala.

```
qsort(s as Seq of Integer) as Seq of Integer
    if s = [] then
        return []
    else
        pivot = Head(s)
        rest = Tail(s)
        return qsort([y | y in rest where y < pivot])
             + [pivot]
             + qsort([y | y in rest where y >= pivot])

Main()
    WriteLine(qsort([1,4,7,3,5]))
```

AsmL and Scala implementations for the quicksort algorithm look very similar. AsmL has the functions `Head` and `Tail` for accessing elements in lists and the operator `+` for concatenating lists.

```
object QuickSort {
  import AsmL._;

  def qsort(a: List[Int]): List[Int] = {
    if (a.isEmpty)
      List()
    else {
      val pivot = a.head;
      val rest = a.tail;
      qsort(for (val y <- rest; y < pivot) yield y)
            ::: List(pivot)
            ::: qsort(for (val y <- rest; y >= pivot) yield y)
    }
  }

  def main(args: Array[String]) =
    Console.println(qsort(List(1, 4, 7, 3 , 5)));
}
```

In the same manner the Scala class `List` defines the functions `head` and `tail` for accessing elements in lists and the operator `:::` for concatenating lists.

6 Variables and State

Both AsmL and Scala programs use variables to specify their state. But in contrast to other programming languages such as C, Java or Scala where changes take place immediately and in sequential order, all changes in AsmL happen simultaneously, when you move from one step to another. Then, all the updates happen at once. This is called an *atomic transaction*.

```
var x = 0

Main()
    step
        WriteLine("In the first step, x = " + x)
    step
        x := 2
        WriteLine("In the second step, x = " + x)
    step
        WriteLine("In the third step, x = " + x)
```

The variable x is updated during the transition from step 2 to step 3.

```
In the first step, x = 0
In the second step, x = 0
In the third step, x = 2
```

Here is an example to compute the lengths of all shortest paths in a graph from a given node s to any node.

```
Shortest(s as Node, // start node
         nodes as Set of Node,
         edges as Map of (Node, Node) to Integer) as Map of Node to Integer
    var S = {s -> 0}
    step until fixpoint
        forall n in nodes where n <> s
            E = {S(m) + edges(m, n) | m in nodes where m in S and (m, n) in edges}
            if Size(E) > 0 then S(n) := min x | x in E
    step
        return S

Main()
    a = Node("A")
    b = Node("B")
    c = Node("C")
    d = Node("D")
    nodes = {a, b, c, d}
    edges = {(a, b) -> 10, (a, c) -> 15, (b, d) -> 12, (c, d) -> 6}

    WriteLine(Shortest(a, nodes, edges))
```

The output result is {C->15, D->21, A->0, B->10} where the token "->" is called the maplet operation in AsmL.

```
object Shortest {

    import AsmL._;
    import scala.collection.mutable._;
```

```
    def shortest(s: Node, // start node
        nodes: Set[Node],
        edges: Map[Pair[Node, Node], Int]): Map[Node, Int] = {

        val s0 = Pair(s, 0);

        def min(xs: List[Int]) = AsmL.min((x: Int, y: Int) => x < y, xs);

        def improve(ss: Map[Node, Int]): Map[Node, Int] = {
            AsmL.Map(s0 :: (for (
              val n <- nodes.toList;
              n != s;
              val dist <- min(for (
                val m <- nodes.toList;
                val d <- ss.get(m).toList;
                val e <- edges.get(Pair(m, n)).toList
              ) yield (d + e)).toList
            ) yield Pair(n, dist)
          ))
        }

        fixpoint(AsmL.Map(s0), improve)
    }

    def main(args: Array[String]) = {
        val a = Node("A"); val b = Node("B");
        val c = Node("C"); val d = Node("D");
        val nodes = AsmL.Set(a, b, c, d);
        val edges = AsmL.Map(
            Pair(Pair(a, b), 10), Pair(Pair(a, c), 15),
            Pair(Pair(b, d), 12), Pair(Pair(c, d),  6));

        Console.println(shortest(a, nodes, edges));
    }

}
```

The functional style is the natural way to express algorithms in SCALA and it allows us to do without any state variable.

7 Structures, Classes and Interfaces

An AsmL structure is similar to a `struct` in C, except it is a pure value: once created, its value is fixed. Thus structures contain constant fields and do not share memory. Like unions in C structures may incorporate case statements as a way of organizing different variant forms.

Classes in AsmL have many similarities to those in other languages. They are templates for user-defined types that can contain both fields and methods. AsmL allows classes and interfaces to be constructed modularly. All definitions are textually concatenated. Each modular piece can contain methods as well as member variables.

```
class Person
    private name as String
    var age as Integer
    public override function ToString() as String?
        return "name=" + name + ", age=" + age

Main()
    paul = new Person("Paul", 25)
    step WriteLine(paul)
    step paul.age := 24
    step WriteLine(paul.age)
```

And here is the same example in Scala.

```
class Person(name: String, _age: Int) {
  var age = _age;
  override def toString() = "name=" + name + ", age=" + age;
}

object Main {
  def main(args: Array[String]) = {
    val paul = new Person("Paul", 25);
    Console.println(paul);
    paul.age = 24;
    Console.println(paul.age)
  }
}
```

In AsmL and Scala an instance of a class is created using the `new` operator in front of the class name and supplying values for the specified fields.

While AsmL and Scala both support single class inheritance, Scala also allows multiple-inheritance of mixins. Methods may be specialized by derived classes using the `override` keyword.

```
interface Expr
    Eval() as Integer

structure ValExpr implements Expr
    val as Integer
```

```
    public Eval() as Integer return val

class AddExpr implements Expr
    lhs as Expr
    rhs as Expr
    public Eval() as Integer return lhs.Eval() + rhs.Eval()

Eval(e as Expr) as Integer
    match e
        ValExpr(v)    : return v
        AddExpr(l, r): return Eval(l) + Eval(r)

Main()
    three = ValExpr(3)
    four  = ValExpr(4)
    seven = new AddExpr(three, four)
    step WriteLine(seven.Eval())
    step WriteLine(Eval(seven))
```

Defining `Eval()` as an instance method implemented by each expression variant is typical for the object-oriented style where behavior is associated with data.

Both AsmL and Scala support patterns to decompose a value into its constituent parts using syntax that mirrors the value's constructor. Patterns are used for *matching*, the process of testing whether the constructor of a given value has the same form as a given pattern, and for *binding*, the process of associating an identifier with a value.

Patterns in AsmL can be literals, identifiers, tuples, enumerations, structures and classes. The universal pattern ("_") can be matched against any value but does not result in a new binding of a name to a value.

Patterns are supported in Scala in a similar way for any types.

This is illustrated in our example by the second `WriteLine` instruction of the program.

```
trait Expr {
  def eval: Int;
}

case class ValExpr(x: Int) extends Expr {
  def eval: Int = x;
}

case class AddExpr(lhs: Expr, rhs: Expr) extends Expr {
  def eval: Int = lhs.eval + rhs.eval;
}

object Exprs {
  def eval(e: Expr): Int = e match {
    case ValExpr(v)     => v
    case AddExpr(l, r) => eval(l) + eval(r)
  }
```

```
  def main(args: Array[String]) = {
    val three = ValExpr(3);
    val four  = ValExpr(4);
    val seven = AddExpr(three, four);
    Console.println(seven.eval);
    Console.println(eval(seven))
  }
}
```

8 Exception Handling

AsmL supports exception handling with `try/catch` expressions. An exception can be generated explicitly using an expression of the form `throw exp`, where `exp` evaluates to a reference of an object that is derived from `System.Exception` class, or it may arise from a runtime event such as a divide-by-zero error. The expression `error exp` can be used to express an unrecoverable error and may be used in any statement context. Errors may not be processed by any exception handler.

We modify here the example presented in listing 1 inorder to handle the thrown exception at runtime.

```
Main()
    directory = {"emergency" -> 911, "info" -> 411}

    step WriteLine(directory("info"))
    step
        try
            WriteLine(directory("hotline"))
        catch
            e: WriteLine("key not found")
```

We now get the following output:

```
411
key not found
```

Exceptions in Scala are handled in a similar manner using the `try/except` expressions. An exception can be generated explicitly using an expression of the form `exp.throw`, where `exp` evaluates to a reference of an object that is derived from `java.lang.RuntimeException` class, or it may arise from a runtime event such as a divide-by-zero error. The expression `error(exp)` can be used to express an unrecoverable error and may be used in any statement context. Unlike in AsmL errors in Scala can be processed by an exception handler.

In a similar manner we modify the example from listing 2:

```
object Composites {

  import AsmL._;
```

```
  def main(args: Array[String]) = {
    val directory = Map(Pair("emergency", 911), Pair("info", 411));

    Console.println(directory("info"));
    try {
      Console.println(directory("hotline"));
    } catch {
      case e: Exception => Console.println("key not found")
    }
  }

}
```

And we get the same output with SCALA:

```
411
key not found
```

Both ASML and SCALA also support user-defined exceptions.

```
class CubeException extends AsmLRuntimeException
    str as String

    CubeException(s as String)
        str = s

    Describe() as String
        return "Cube Exception: " + str

structure Side
    teeth as Seq of Boolean
    Side(ts as Seq of Boolean)
        teeth = side_constraint(ts)
    shared side_constraint(bs as Seq of Boolean) as Seq of Boolean
        if bs.Length gt 2
            return bs
        else
            throw new CubeException("Side constraint violated")

Main()
    step WriteLine(Side([true, true, false, true]))
    step WriteLine(Side([true, true]))
```

The second instruction in `Main()` generates the following exception:

```
Side(teath=[True, True, False, True])

Unhandled Exception: Application.CubeException: Exception in type
Application.CubeException was thrown.
  at Application.Side.side_constraint_Boolean(Seq bs) in ...\Exceptions.asml:line 28
  at Application._GLOBAL_.Main() in ...\Exceptions.asml:line 32
```

And here is the same example written in SCALA:

```
class CubeException(s: String) extends java.lang.RuntimeException(s) {
  override def getMessage() = "Cube Exception: " + s;
}

object Side0 {
  def side_constraint(bs: List[Boolean]) =
    if (bs.length < 3) throw new CubeException("Side constraint violated");
}

case class Side(ts: List[Boolean]) {
  Side0.side_constraint(ts);
}

object Exceptions {
  def main(args: Array[String]) = {
    Console.println(Side(List(true, true, false, true)));
    Console.println(Side(List(true, true)))
  }
}
```

Again we observe a similar behavior in SCALA:

```
Side(List(true,true,false,true))
Exception in thread "main" CubeException$class: Cube Exception: Side constraint violated
        at Side0$.side_constraint(src/Exceptions.scala:18)
        at Side$class.<init>(src/Exceptions.scala:22)
        at Exceptions$.main(src/Exceptions.scala:29)
        at Exceptions.main(src/Exceptions.scala)
```

9 Generics

AsmL and Scala both support *generics*. Generics - also commonly known as *parametrized types* - are useful because many common classes and interfaces can be parametrized by the type of the data being stored and manipulated. Methods may also be parametrized by type inorder to implement ”generic algorithms”.

In the example below we create a `Stack` generic class declaration where we specify a type parameter, called `T`, using the `of` keyword.

```
class Stack of T
    private var s as Seq of T = []
    Push(data as T) s := [data] + s
    Pop() as T
        if Size(s) <= 0 then
            error "Pop(): empty stack error"
        else
            head = Head(s)
            s := Tail(s)
            return head
    Top() as T
        if Size(s) <= 0 then
            error "Top(): empty stack error"
        else
            return Head(s)
    Size() as Integer return Size(s)
    public override ToString() as String? return s.ToString()

PushMultiple of T (stack as Stack of T, s as Seq of T)
    step foreach x in s
        stack.Push(x)

Main()
    stack = new Stack of Integer
    step stack.Push(3)
    step stack.Push(1)
    step PushMultiple(stack, [2, 5])
    step WriteLine("stack = " + stack)
    step WriteLine("stack.Pop() = " + stack.Pop())
    step WriteLine("stack.Top() = " + stack.Top())
    step WriteLine("stack.Size() = " + stack.Size())
```

The output result is:

```
stack = [5, 2, 1, 3]
stack.Pop() = 5
stack.Top() = 2
stack.Size() = 3
```

The declaration of the constant object `stack` in the `Main()` function is the only place where we need to specify an explicit type parameter.

In Scala the type parameter list - which consists of one parameter `a` in our example - is declared in square brackets and follows the class name in a class declaration (idem for generic function declarations):

```
class Stack[T] {
  private var s: List[T] = List();
  def push(data: T) = s = data :: s;
  def pop = s match {
    case Nil    => error("pop: empty stack error")
    case h :: t => s = t; h
  }
  def top = s match {
    case Nil    => error("top: empty stack error")
    case h :: t => h
  }
  def size = s.length;
  override def toString() = s.mkString("[", ", ", "]");
}

object Main {
  def pushMultiple[T](stack: Stack[T], s: List[T]) =
    s foreach { x => stack.push(x) };

  def main(args: Array[String]) = {
    val stack = new Stack[Int];
    stack.push(3);
    stack push 1; // infix notation
    pushMultiple(stack, List(2, 5));
    Console.println("stack = " + stack);
    Console.println("stack.pop = " + stack.pop);
    Console.println("stack.top = " + stack.top);
    Console.println("stack.size = " + stack.size)
  }
}
```

As practical advantages generics increase the programmer productivity by promoting code reuse, improve code robustness by allowing static type-checking and increase program performance by reducing needed checks at runtime.

10 Conclusion

AsmL and Scala are both strongly-typed languages. Scala is a concise language, its syntax is small (37 vs. 49 keywords in Java [6] vs. 76 in C#) unlike AsmL which contains about three times more keywords. Scala is a general-purpose language and it's not a surprise that AsmL code is more compact than Scala code for domain-specific applications.

The integration of each language with a mainstream programming environment - Microsoft's Clr for AsmL and Sun's Jvm for Scala should increase its applicability and acceptance; in particular it makes possible to develop hybrid applications that blend different programming paradigms into a single, cohesive unit.

One important point about programming languages is that stability, performance, and comprehensiveness of the library contributes as much to overall productivity as the language itself. AsmL and Scala are still under development and Scala has not yet reached the same level of maturity as AsmL, in particular concerning stability and performance.

The strengths of the AsmL are:

- The AsmL language is developed by the same people who made the ASM specification. In particular the non-determinism of AsmL contrasts to the sequential execution model used by most programming languages including Scala.
- Programming in AsmL is served by powerful development tools. The AsmL compiler accepts both C# and AsmL source files and provides Xml/Word integration. The AsmL Test Generator tool is an integrated test generation environment. It can be used to automatically generate test cases from an AsmL model using various algorithms, and to use such test cases to perform a conformance test against an actual implementation.

The strengths of the Scala are:

- Scala is a multi-paradigm programming language designed to express common programming patterns in a consise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages.
- Scala provides a unique combination of language mechanisms that make it easy to smoothly add new language constructs in form of libraries (i.e. any method may be used as an infix or postfix operator and closures are constructed automatically depending on the expected type).

For the sake of compactness this paper does not cover some interesting feature of AsmL and Scala such as type systems (i.e. type inference, abstract types and covariance of collection types) and support for concurrency.

As a complement to this brief comparison between AsmL and Scala we present some key features of the two languages in the appendix (Table 5).

Acknowledgments

I am grateful to Martin Odersky, Matthias Zenger and Erik Stenman for valuable comments on previous drafts of this paper and to Vincent Cremet for his contribution to the code examples. This paper was motivated in part by a discussion led by Prof. Alain Wegmann, Prof. Martin Odersky and myself on SCALA as a possible alternative to the ASML language for programming ASM models.

References

[1] Ergon Börger. *The Origins and the Development of the ASM Method for High Level System Design and Analysis*, 2002. `http://www.di.unipi.it/~boerger/`.

[2] Microsoft Corp. *AsmL: The Abstract State Machine Language*, 2002. `http://research.microsoft.com/fse/asml/`.

[3] Microsoft Corp. *Introducing AsmL: A Tutorial for the Abstract State Machine Language*, 2002. `http://research.microsoft.com/fse/asml/`.

[4] Microsoft Corp. *AsmL 2 Release Notes*, 2003. `http://research.microsoft.com/fse/asml/`.

[5] Yuri Gurevich. *May 1997 Draft of the ASM Guide*, 1997. `http://research.microsoft.com/~gurevich/`.

[6] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha. *The Java Language Specification*, 2000. `http://java.sun.com/docs/books/jls/`.

[7] Martin Odersky and al. *Scala Language Specification*, 2004. `http://scala.epfl.ch/`.

[8] Foundations of Software Engineering (FSE). *An Introduction to ASML 1.5*, 2002.

[9] Igor Potapov. *Software Engineering Course - Lectures 6-12*, 2001. `http://www.csc.liv.ac.uk/~igor/COMP201/`.

[10] Matthias Zenger. *Extensible Algebraic Datatypes with Defaults*, 2001. `http://lampwww.epfl.ch/~zenger/jaco/`.

Appendix

abstract	enumerated	if	mod	property	subsetseq
add	eq (=)	iff	mybase	process	superset
allpublic	event	ifnone	namespace	protected	supersetseq
and	exists	implements	new	public	the
any	explore	implies	nonvirtual	ref	then
as	extends	import	not	remove	throw
case	false	in	ne (<>)	require	to
catch	fixpoint	initially	notin	resulting	true
choose	for	inout	null	return	try
class	forall	interface	of	sealead	type
const	foreach	internal	operator	search	undef
constraint	foreign	intersect	or	separate	union
delegate	friend	is	otherwise	set	unique
difference	from	lt (<)	out	shadows	until
do	function	lte (<=)	override	shared	var
else	get	let	power	skip	virtual
elseif	gt (>)	match	private	step	where
ensure	gte (>=)	me	primitive	structure	while
enum	holds	merge	procedure	subset	with

Table 1: AsmL keywords (114)

abstract	else	if	object	sealed	try
case	extends	implicit	override	super	type
catch	false	import	package	this	val
class	final	match	private	throw	var
def	finally	new	protected	trait	while
do	for	null	return	true	with
					yield

Table 2: Scala keywords (37)

AsmL	**Scala**	**Description**
Boolean	Boolean	values true and false
Null	AllRef	value null
Byte	Byte	8-bit unsigned integer type
Short	Short	16-bit signed integer type
Integer	Int	32-bit signed integer type
Long	Long	64-bit signed integer type
Float	Float	32-bit floating-point type
Double	Double	64-bit floating-point type
Char	Char	Unicode character type
String	String	Unicode string type
-	Symbol	symbolic names (i.e. XML tags)
Void	Unit	statements

Table 3: Predefined primitive types

AsmL	Scala	Description
`Seq of A`	`Seq[A]`, `List[A]`	all sequences containing elements of type A
`Set of A`	`Set[A]`	all sets containing elements of type A
`Map of A to B`	`Map[A, B]`	all tables whose entries map elements of type A to element of type B
`(A1,A2,..)`	`Tuple(A1,A2,..)`	all tuples consisting of elements of type A1, A2, .. with the two shortcuts `Pair[A,B]` and `Triple[A,B,C]`
`A?`	`Option[A]`	all the values of type A plus the special value `null` in AsmL and `None` in Scala
-	`Array[A]`	all arrays containing elements of type A
-	`Iterator[A]`	all iterators on elements of type A
-	`Stream[A]`	all lazy sequences on elements of type A

Table 4: Predefined composite types

Feature	AsmL	Scala
variable declaration	global/local/instance	-/local/instance
function declaration	global/-/instance	-/local/instance
nested declarations	-	✓
higher-order functions	-	✓
type inference	✓	✓
type parametrization	✓	✓
type covariance	✓	✓
class inheritance	✓	✓
class constructor	implicit/user-defined	implicit/user-defined
method overloading	✓	✓
method overriding	`override`	`override`
pattern matching	✓	✓
exception handling	`try/catch/-`	`try/catch/finally`
namespace management	`namespace`	`package`
import aliasing	`=`	`=>`
compiler directives	✓[1]	-

Table 5: Short feature comparison

[1] i.e. `[AsmL.Profile(Accesses=true)]` means that all accesses to fields get profiled.