Published: Proc. of the Joint Modular Languages Conference, JMLC'97, Linz, Austria, March, 1997.

On Extending Java

Andreas Krall[1] Jan Vitek[2]

[1] Institut für Computersprachen, Technische Universität Wien, Argentinierstr. 8, A-1040 Wien, Austria, e-mail: andi@complang.tuwien.ac.at web: http://www.complang.tuwien.ac.at/andi/

[2] Object Systems Group, Centre Universitaire d'Informatique, Uni. of Geneva, Geneva, Switzerland. e-mail: jvitek@cui.unige.ch web: http://cuiwww.unige.ch/~jvitek/

Abstract. The design of Java sports a simple and elegant object model. Its simplicity may well be the language's main selling point—it is both easy to learn and to implement—but in the long run the same simplicity may prove to be a sign of a lack of expressive power that could hinder the development of large software systems. We present four non-intrusive language extensions, *tuples*, *closures*, *anonymous objects* and *iterators*, give examples of use and detail a translation scheme into plain Java. These extensions enhance the expressive power of Java and allow certain common programming idioms to be coded more naturally.

1 Introduction

The role of high-level programming languages is to lay a veneer of abstraction over the bare machine in order to provide software developers with the means of expressing algorithms. The constructs of a language—loops, routines and such—abstract over frequently repeated patterns of machine code, reflecting and fostering a certain coding style. Java is a new object-oriented programming language remarkable for its conceptual simplicity: everything revolves around classes. The class is the only mechanism for defining new abstractions available to the programmer. Thus every abstraction must be expressed in terms of classes, even when other abstraction mechanisms would be better suited. We feel that Java suffers from a heavy handed application of Occam's razor to the extent that some simple tasks and common programming idioms require cumbersome and error prone coding. It is a view shared by other researchers, if the number of proposals to patch up the language is any indication [3][13][14][15].

This paper presents and details four non-intrusive extensions of the Java language. By non-intrusive we mean that they preserve the semantics of Java, *i.e.* the meaning of existing code is not affected. For our implementation we had two choices: either generate byte codes, or perform a source-to-source translation. The Java VM is so tightly coupled to the source language [13] that there is little real difference between the two. For the sake of clarity we present our extensions as source program transformations. The extensions, *tuples*, *closures*, *anonymous object*s and *iterators*, are well known programming language features which we adapt here to fit Java. Our contribution is twofold. First we show how Java can be extended, providing a discussion which can be useful to implementors wishing to modify the language or use it as a portable intermediate language. Second, our tuples and iterators differ from existing proposals and we present an efficient translation scheme. Clearly, these extensions do not purport to solve all Java's problems, far from it. Important contributions have already addressed the issue of bounded parametric polymorphism [3][13] and nested classes [15], and further research is needed.

This work is an offshoot of research on mobile computations carried out within the Swiss FNRS ASAP project. We are implementing a new object-oriented language, named SEAL, that allows running computations to move between different hosts on the net. Java is the intermediate language of the SEAL compiler and the extensions we present are features of SEAL. The translation schemes are close renditions of the translation schemes from SEAL to Java. There is a long tradition of using high level languages as portable intermediate languages. The usual choice is C, but there are arguments in favour of Java. A strongly typed language eases the task of developing a code generator as the output of the compiler is easier to read and understand. Garbage collection and support for object-oriented programming are already there, the SEAL compiler may use them as such. Moreover, Java brings byte-code portability and standardized user interface libraries. Java's drawback is that if a straightforward mapping from SEAL constructs to Java constructs can not be found then the semantics must be emulated in software, *e.g.* [7] emulates a Nesl VM on top of the Java VM, where each Nesl VM instruction corresponds to a function written in Java. Lower level constructs pose the most problems. For instance, it would be difficult to translate a language with unrestricted memory access and explicit memory management. Nevertheless, Java has already been used as a target for languages such as Ada [17], Scheme [4], Basic [9], Nesl [7]. The work on SEAL is ongoing.

2 Language Extensions

We describe four Java language extensions designed to support a simple translation scheme to plain Java or to byte code. For each extension, we sketch semantics and usage, describe the key points of the translation scheme and conclude with a discussion. For clarity, the translation to plain Java is described below. During the translation process new class names have to be generated; we simply chose to extend user defined names by sequential integer values. If these conflict with existing names, the user can specify a prefix to the translator-generated names [15].

We proceed with a brief description of Java [12]. A class is a template for creating objects, or *instances*. It defines data attributes, *instance variables*, and operations, *methods*, of its instances. It is also a run-time repository for shared data and operations called static variables and methods[1]. Each class implicitly defines a homonymous type. Classes may inherit from other classes, thus extending the set of data attributes and methods of the objects they define. Java only allows single-inheritance, that is, a class may *extend* a single parent. Interfaces are named sets of method signatures which implicitly define homonymous types. A class may implement a number of interfaces, *i.e* provide implementations for all methods of these interfaces. Interfaces introduce multiple subtyping. Java has an exception mechanism. A statement which raises an exception causes a non-local control flow branch to the first handler of this type of exception. Exceptions follow the termination model. These features are not new, they can be traced to C for the syntax, Smalltalk-76 for the object model, Modula-3 for the exceptions, C++ for constructors and static typing. As always, it is interesting to study what was left out of a design and ponder why. For instance, C got around its lack of high level abstraction mechanism with a very liberal type system and the ability to call arbitrary code through function pointers. In Smalltalk, unnamed procedures, called blocks, are used through-

1. The use of the keyword "static" is somewhat confusing. It refers to the fact that instances of a class are created dynamically while the class itself is created and initialized at load-time.

out the libraries. Both are missing in Java. Statically typed languages, on the other hand, have difficulties typing container classes. C++ solves the problem by introducing parameterized types. Java does not have them, and forces programmers to rely on run-time typechecking or to write similar code many times. Java lacks an explicit type declaration statement, the designers must have felt that it was redundant, they chose to include interfaces instead. But interfaces are restricted as they can not contain data attributes. The reason for this choice is pragmatic, adding data attributes would complicate the language implementor's life by recreating the problems that stem from multiple inheritance. These omissions and restrictions motivate this paper.

3 The First New Construct: Tuples

Tuples are typed sequences of values that can be viewed either as special kinds of arrays or as simplified objects. They have been given a lightweight syntax that makes them useful to represent typed heterogeneous containers and to implement multiple return values for functions. A tuple type is a cartesian product of types, each type in the product corresponds to the type of one of the tuple elements. Thus, a tuple is both polymorphic, by inclusion on the element types, and heterogeneous because the types of elements can differ. The type $[t_0, ..., t_n]$ denotes a n-tuple where t_i is the type of the i-th element. A tuple type declaration in the extended language has the form

type T = $[t_0, ..., t_n]$;

which introduces a type T describing n-tuples of type $[t_0, ..., t_n]$. The expression [3, true] creates a tuple literal with two fields containing the integer 3 and the boolean true respectively. The types of tuple variables and tuple valued functions must be specified, whereas the type of a literal is inferred from context. The following code fragment illustrates the use and syntax of tuple operations:

```
type Pair = [int, boolean];     // Tuple type declaration
Pair p, s;                      // Tuple variables
int i; boolean b;
p  =  [3, true];                // Creation of a tuple and assignment to a variable.
s  =  p;                        // Tuple value assignment
s[0] =1;                        // Element assignment
i  =  s[0];                     // Element extraction
[i, b] = p;                     // Element extraction
[i, b] = id([i, b]);            // Type of function id is Pair id(Pair)
[i, b] = id(p);
return [2, b];                  // Returning a newly created tuple
```

Tuple values are always transmitted by copy, thus a function invocation such as id(s) creates a copy of the tuple. The tuple variable assignment s = p performs an element-wise copy. The equality test is implemented as an element-wise comparison. All tuple variables are initialized to empty tuples, so the declaration Pair p; is equivalent to Pair p = [null, null];. Different tuple types are not related by subtyping. Tuple types are to be primitive types and are not subtypes of Object, they can not be stored in variables of another type.

We mentioned that tuples can be viewed as either arrays or object. In the array view, tuples are typed heterogeneous arrays of fixed sized for which the arity appears in the type. Element extraction is syntactic sugar for array access. Tuples differ from plain arrays because they allow individual elements to have different types yet remain type safe. They tend to remain small as it is necessary to declare the type of each element. In the object view, tuples are objects with unnamed fields and no user defined code. Consider a function that negates two values and returns the pair as a tuple:

```
type Pair = [Integer, Boolean];                    // Tuple type definition
Pair negate(Integer i, Boolean b) {                // Function definition
    return [-i, not b];
}

[val, truth] = negate(42,true)                     // Function invocation
```

Using arrays, the same function must return an array of objects. Array accesses will require bound checking as well as dynamic typechecking. The result is less efficient and, in our opinion, harder to understand.

```
Object[] negate(Integer i; Boolean b) {
    Object[] o = {-i,not b};
    return o;
}
Object[] temp = negate(42, true);
val = (Integer)temp[0];
truth = (Boolean)temp[1];
```

The other alternative is to define a new object class. This version must bear the additional cost of allocating the return object on the heap. If the compiler is not able to inline the constructor call, an extra function call must be issued. The solution is definitely more verbose.

```
class Pair {
    public Integer first;
    public Boolean second;
    Pair (Integer i, Boolean b) {
        first = i; second = b;
    }
}
Pair negate(Integer i; Boolean b) {
    return new Pair(-i,not b);
}
Pair temp = negate(42,true);
val = temp.first;
truth = temp.second;
```

Translation In Java, types are introduced by class definitions. Therefore tuple type declarations must be translated into class definitions. Tuples map to classes with translator-generated field names. In addition, a constructor must be added to complete the class definition.

<u>source</u>	<u>translation</u>

`type NameKey = [String, int];`	`final class NameKey {`
	`    String tuple0; int tuple1;`
	`    NameKey(String t0, int t1) {`
	`        tuple0 = t0;`
	`        tuple1 = t1;`
	`    }`
	`}`

A tuple can be used as an expression (literal) and on the left hand side of an assignment (lvalue). Tuple literals are translated to constructor calls /*1*/. For lvalues, fields are extracted from the rvalue and assigned to the lvalue from left to right /*2*/. If both sides of an assignment are tuples, the constructor call is eliminated and the assignment is split up /*3*/. Indexed access is translated to a field access /*4*/.

```
class tuple {
   public static void main(String args[]) {
      NameKey nk, nk1; String name; int key;
      nk = ["andi", 1];                        /*1*/
      [name, key] = nk;                        /*2*/
      [name, key] = ["andi", 1];               /*3*/
      name = nk[0];                            /*4*/
      nk[1] = 1;                               /*4*/
      nk1 = nk;
   }
}
            source
```

```
class tuple {
   public static void main(String args[]) {
      NameKey nk, nk1; String name; int key;
      nk = new NameKey("andi", 1);             /*1*/
      name = nk.tuple0; key = nk.tuple1;       /*2*/
      name = "andi"; key = 1;                  /*3*/
      name = nk.tuple0;                        /*4*/
      nk.tuple1 = 1;                           /*4*/
      nk1 = new NameKey(nk.tuple0, nk.tuple1);
   }
}
                     translation
```

Discussion Adding tuples to Java does not make the language significantly more complex and does not require changes to the virtual machine. Their usefulness stems from their syntactic convenience and from their amenability to compiler optimizations. The fact that tuple types are not subtypes of Object guarantees that tuples can not be stored in heterogeneous data structures. This together with copy semantics prevents tuples from being aliased creating the opportunity for the compiler to avoid constructing tuples altogether. For tuple valued functions, tuples can be passed on the stack. The usefulness of tuples increases if they are coupled with pattern matching. A number of alternatives for a syntax and semantics of pattern matching are being considered.

Non-intrusive extensions are always limited by the language being extended. In the case of tuples, we would have preferred to use the same syntax as array initializers for tuple literals, *i.e.* curly braces, but this would complicate parsing. The second problem is that subtyping between tuple types is not possible, at least not with our current translation scheme. We would like to be able to base subtyping of tuples on structural subtyping. For instance, a tuple type (the child) is a subtype of another tuple type (the parent) if both have the same arity and if the type of each field of the child is a subtype of the type of the corresponding field in the parent. Structural subtyping conflicts with Java's otherwise explicit name based subtyping.

4 The Second New Construct: Closures

Closures are functions defined inside the scope of an object or method and closed over their free lexical variables. A closure is a value which can be stored, copied, given as an argument and invoked to yield a result. Closures allow a higher order programming style in which the programmer writes functions to manipulate other functions. Functional languages and languages such as Smalltalk-80 and Beta support closures. In fact, higher order functions are already implicitly present in objects. Having explicit closures simplifies the coding of many common tasks; two examples of which are callbacks in graphical user interfaces and manipulation of the elements of container data structures. The following expression elaborates into a closure:

```
SimpleFun(int y){ return x + y + z + w; }
```

SimpleFun is a closure type followed by an argument list and the suspended function body. The body of the closure refers to three free variables, x, y and z, which can either be local variables of an enclosing function or method, members of the enclosing object referred to by this, or static members of the enclosing object's class.

```
type SimpleFun = (int) -> void;                       // Unnamed function type declaration
class P {                                             // Class definition
  private int x;
  static int y;
  public SimpleFun adder(int z) {
    return SimpleFun(int w){ return x + y + z + w; };          //Unnamed function creation
  }
```

In this example x is member of the enclosing object, y is a static member and z is a local argument of the enclosing function. The function is closed over these free variables. The implementation must therefore ensure that their lifetimes exceed that of the closure. Invocation is as expected:

```
SimpleFun s = p.adder(4);                             // p is of class P
x = s(4)* 3;                                          // the closure is invoked
```

Translation A closure type is translated to an abstract class with a single abstract function named eval which has the same signature as the closure type. Each closure of this type is translated to a class which extends the abstract class. The body of the closure translates into the body of eval. A definition of the closure is translated into an instantiation, and an invocation of the closure is replaced by a call of eval.

source	translation

`type VoidObjectFun = (Object o) -> void;`	`abstract class VoidObjectFun`
	`  abstract void eval(Object o);`
	`}`

Since closures can access variables of the enclosing scope, we must consider visibility and lifetime of variables. Variables must be visible from the new class generated by the translator and they must not be garbage collected before the end of the closure's execution. *For local variables of the enclosing function*: There are two categories of variables: read-only and read-write variables. The former can be passed by value as additional arguments to the function. The latter have to be passed by reference. Passing by reference requires encapsulation of the variable in an object. We create one object, called an *environment*, for all variables that may be modified by the closure /*1*/. The enclosing function must create the environment /*2*/ and initialize it with the values of the local variables. Arguments of the enclosing function are treated as local variables. The enclosing function is modified so that accesses to variables used by the closure become accesses to the environment /*3*/. The closure class must include an instance variable that will point to the environment /*4*/. Determining which variables will be modified requires static analysis of the closure body. In general, it is not possible to be exact. A safe approximation is to include all variables that appear in the closure's body. *For instance variables of the enclosing object*: The translator ensures that the closure will have a reference to the enclosing object, this reference is stored in an instance variable of the closure class /*5*/. The two reference fields are initialized by a constructor of the closure class /*6*/. Fields of the enclosing object are accessed via outobj /*7*/ and fields of the enclosing function via outenv /*8*/. There is a further visibility issue if instance variables of the enclosing object have been declared as private. In this case the translator changes the visibility of the corresponding instance variables from private to the default. This means that other classes in the same package may see these variables. The translator is responsible of checking that other classes do not try to access the instance variables which have just been made visible. Safety may be compromised if another file is compiled separately within the same package. The definition of the clo-

```
class ObjSeq {
    Object seq[];
    ObjSeq(int size) {
        seq = new Object[size];
    }
    void apply(VoidObjectFun fun) {
        for (int i = 0; i < seq.length; i++) {
            fun(seq[i]);
}   }   }

class high {
    String str = "High: ";
    public void test(String args[]) {

        String s = "Seq: ";                  /*1,3*/
        ObjSeq os = new ObjSeq(5);
        os.apply(VoidObjectFun (obj) {
          System.out.
                  println(str+s+obj.toString());
        } );
}   }
```

source

```
final class Env0 {
    String s;                                          /*1*/
}
final class VoidObjectFun0
                        extends VoidObjectFun {
    private Env0 outenv;                               /*4*/
    private high outobj;                               /*5*/
    VoidObjectFun0(high o, Env0 e) {                   /*6*/
        outobj = o; outenv = e;
    }
    void eval(Object obj) {
        System.out.println(this.outobj.str             /*7*/
        +this.outenv.s+obj.toString());                /*8*/
}   }
class ObjSeq {
    Object seq[];
        ObjSeq(int size) {
        seq = new Object[size];
    }
    void apply(VoidObjectFun fun) {
        for (int i = 0; i < seq.length; i++) {
            fun.eval(seq[i]);                         /*10*/
}   }   }
class high {
    String str = "High: ";
    public void test(String args[]) {
        Env0 env = new Env0();                         /*2*/
        env.s = "Seq: ";                               /*3*/
        ObjSeq os = new ObjSeq(5);
        os.apply(
            new VoidObjectFun0(this, env)              /*9*/
        );
}   }
```

translation

sure is replaced by a constructor call with two arguments: the object (this) and the environment /*9*/. Closure invocation is replaced by a call to eval /*10*/.

Discussion Closures are popular. Since this paper was submitted two implementations of closures have been described [13] and [14]. The concept of higher order functions in Pizza is similar to ours but the translation to Java is quite different. In Pizza every variable of an enclosing function is passed as an additional function argument. Variables which are modified have to be passed as a reference argument. This is accomplished by putting the variable in a single element array. (The same translation scheme for accessing variables of an enclosing function is used for the translation of inner classes [15].) This leads to the creation of a large number of single element arrays and to functions with very large argument lists. The overhead of heap allocating all of these one-element arrays may be a performance bottleneck. In our translation scheme, we create a single environment object for all variables of an enclosing function and pass only one additional argument to the closure.

Our implementation is similar to closures in functional languages [8] and nested scoping in Pascal and Modula-2 [2]. In a machine level implementation, a pointer to the stack frame would replace the reference to the environment and give faster access to the variables. Our implementation has the advantage that only variables of the environment have to be put on the heap, whereas in a machine level implementation the complete stack frame must be put on the heap. For faster variable access, the translator generates a separate environment object only if more than one anonymous function is used inside a method. Otherwise the function object contains the variables of the environment instead of a reference to the environment. An environment or a reference to the object is generated only if enclosing variables are accessed. All implementations of higher order functions require the presence of a garbage collector. Otherwise it would be difficult to reclaim environment objects.

5 The Third New Construct: Anonymous Objects

Java offers mechanisms for writing many of its data types literally, e.g. integers, strings, arrays, classes. Unlike Modula-3 and Self, Java has no support for expressing literal objects. An anonymous object is an expression which extends an existing class and yields an instance of this anonymous class. The anonymous object is nested within some method definition and has the same access to variables as an unnamed function. The structure of an anonymous object is

```
<BaseClass> { ... <Extension> ... } ( ... <Constructor arguments> ... )
```

An example of an extension is:

```
BigNum{
        int i = x;                                                          /*1*/
        public BigNum add(BigNum n) {
            System.out.println("Add" + toString() + " to " + n.toString());
            return super.add(n);
        }
    }(3);
```

This yields an object of an anonymous subclass of BigNum with a new instance variable i and a new implementation of the method add(BigNum). Note the reference to a free variable x in the assignment to i /*1*/. A valid context for this expression could be:

```
class BigNum {
    public BigNum add(BigNum n) { ... }
    public BigNum(int i) { ... }
}
class P {
    public BigNum foo() {
        int x = ...;
        return BigNum{
            int i = x;
            public BigNum add(BigNum n) {
                System.out.println("Add" + toString() + " to " + n.toString());
                return super.add(n);
            }
        }(3);
    }
}
```

We have chosen to grant access to all variables and methods of the lexically enclosing object to an anonymous object, because we want the freedom afforded by friend declarations in C++ which proves indispensable for the efficient iterators of section 6. Anonymous objects are not allowed to define static variables or methods as it would not be clear when to initialize them, especially if the initializer was defined in terms of the dynamic values of enclosing variables.

Translation An anonymous object is translated to a new class definition. Accesses to variables declared in the enclosing function or to fields of an enclosing object are translated in the same way as closures. The only difference is that the new class cannot be used as an environment as in the optimization for closures. The constructor for the extended class gets two additional arguments for the enclosing object and environment. These two fields are initialized after the constructor of the super class has been called.

```
class Xtd {
    int x;
    public Xtd(int i) {x = i;}
}

class B {
    int i_B = 1;
    Xtd fB() {
        int i_fB = 2;
        return Xtd {
            int ext = x + i_B + i_fB;
        } (5);
    }
}
```

source

```
class Xtd {
    int x;
    public Xtd(int i) {x = i;}
}
final class Xtd000 extends Xtd {
    private B outobj;
    private Env000_BfB outenv;
    int ext;
    Xtd000(B o, Env000_BfB e) {
        super(5);
        outobj = o;
        outenv = e;
        ext = x + outobj.i_B + outenv.i_fB;
    }
}
final class Env000_BfB {
    int i_fB;
}
class B {
    int i_B;
    Xtd fB() {
        Env000_BfB env = new Env000_BfB();
        env.i_fB = 2;
        return new Xtd000(this, env);
    }
}
```

translation

Discussion The functionality of anonymous objects is subsumed by a new feature of Java called inner classes [15]. The details of their design were published a couple of months after this paper had been written and submitted; they generalize the concept of anonymous object and allow class definitions to nest. Our translation scheme differs from the Sun proposal as described in the previous section.

6 The Fourth New Construct: Iterators

Iteration over data structures is a common and repetitive task that often requires intimate knowledge of the implementation of the data structures on which the iteration is being carried out. Empirical evidence indicates that iteration code is tricky as errors related to initialization and stopping conditions are frequent. Encapsulating iteration protocols, thus shifting responsibility from client to provider, has proven quite effective in reducing coding effort and improving reliability. For instance, iteration-intensive classes in the Sather [11] library have been reduced to a third of their size by the addition of explicit support for iteration.

An *iterator* is a control abstraction which encapsulates a particular traversal order over a container object. Examples of containers include lists, arrays, matrices, hash tables, sets, or even sequences of integers. Iterators are used in loop constructs to return elements of the container one by one. Before discussing how to design iterators, let's review some requirements for a general purpose iteration abstraction:

- **interface**: the interface should allow one to retrieve, modify, and, possibly, add or delete, elements of the target data structure without having to understand the implementation of the underlying abstraction,
- **multiplicity**: multiple iterators may iterate over the same data structure; the same iterator must be usable in different contexts without losing track of its position in the data structure,
- **composability**: multiple data-structures may be iterated over in a single loop; simple iterators may be composed to form more complex ones,
- **efficiency**: the code of the iterator must have direct access to the data structure over which it iterates; the compiler should be able to compile away the iterator abstraction.

With no additional support from the language, iteration protocols may be encapsulated in objects. Thus, for example, a `Tree` object may have a method `inOrder()` that returns an `InOrderTreeIter`—an object with an interface for in-order tree traversal and methods for checking if the complete tree has been visited. The same class could have other methods returning pre- and post-order iterators. This solution meets the first three requirements. As for efficiency, the problem is that the iterator objects require privileged access to the container object (as with C++'s `friend` declaration) and that they rely heavily on the compiler to optimize away the dynamic nature of allocation and method invocation.

Alternatively, the iterator can be made part of the container object. Thus, for instance, a `Tree` object might have a method `initIter()` which would take three arguments: the order of traversal, a method `next()` which would return the next element and a method `done()` that would return `true` if the entire tree had already been visited. The advantage of this solution is that the iteration code has access to the implementation of the data abstraction. The disadvantage is that it is not easy to implement multiple iterators over the same object, as it would be necessary to keep track of the source of each invocation and maintain a stack of states for all active iterators. The designers of CLU found a solution to both problems. In CLU an iterator is an operation (i.e. method) of an abstract data type. An invocation of an iterator has the general form:

```
for <loop variable> in <iterator invocation> do
   <body> end;
```

Semantically an iterator is a coroutine, it may thus yield a value (this value is bound to the loop variable and used within the loop body), or terminate the loop by returning. There are four major weaknesses of CLU's iterators: (1) the restriction to one iterator per loop, *i.e.* it is not possible to iterate over two data structures at the same time, (2) the inability to modify the data structure—CLU iterators are uni-directional, information flows from the iterator to the loop body, and never the other way around— (3) the lack of composability—CLU iterator abstractions can not be composed or abstracted further—and (4) there is no provision for keeping the state of an iterator between loops, *i.e.* to traverse part of the tree in one loop and the other part in another loop.

Sather generalizes CLU iterators. Iterators are still coroutines, but they can be used anywhere inside the loop body and some of their arguments are marked as "hot" to mean that they are re-evaluted each time control is passed back to the iterator. An iterator is thus a special kind of method implemented as a co-routine which can either yield a value, with the keyword yield, or terminate the loop with the keyword quit. Sather puts the iterator in charge of checking for termination and breaking loops. This clever idea simplifies the task of writing code that iterates over multiple data structures. Hot arguments may be used to modify the data structure. For example, consider the task of copying an array. Assuming that arr1 and arr2 are isomorphic arrays, the Sather code for copying the contents, starting at index i, of arr1 into arr2 is:

```
loop
   v := arr1.next!(i);
   arr2.set!(i, v);
end
```

Here i is the starting index, next!() and set!() are iterator invocations; either one could terminate the loop if the end of the corresponding array is reached. The second argument of set!() needs to be hot in order to provide a new v each time control is transferred to set!(). One drawback of hot arguments is that in order to know which arguments are hot, one must look at the class interface. Each syntactic occurrence of an iterator denotes a different co-routine; in the following Sather code, the values of v1 and v2 are always equal:

```
loop
   v1 := arr1.next!(i);
   v2 := arr1.next!(i);
end
```

We adopt a simpler and more elegant model for iteration in Java. Iterators are objects which have one or more iteration methods. We add a single keyword “iter” which is used to qualify iteration methods. An iteration method is a method which may either: (1) return a value, using the normal return keyword, (2) terminate a loop using the break keyword or (3) go to the loop head, using the continue keyword. An iterator which returns a bounded sequence of positive odd numbers would be coded:

```
class OddIntIter {
   int i = -1;

   int Max;

   iter int next() {
      if (i >= Max)
         break next;
      else {
         i = i + 2;
         return i;
      }
   }

   public OddIntIter(int m) {
      Max = m;
   }
}
```

Note that break and continue can be labelled by the name of the enclosing iter to indicate that control flow should leave the iterator breaking any inner loops. An example of the use of an iterator to sum the odd numbers between 0 and 20 is as follows:

```
o = OddIntIter(20);
while(true)
   x += o.next();
```

We provide the keyword **loop** as syntactic sugar for **while**(true). An advantage of our approach is that the iterator is an entity which has a denotation, thus in the following code fragment v1 and v2 actually denote two consecutive odd integers (which would not be possible with Sather iterators):

```
loop {
    v1 = o.next();
    v2 = o.next();
}
```

We avoid the semantic complication of having to deal with hot arguments and coroutines. Iterators are objects like any other and calls to iterators are normal method invocations. Using anonymous objects allows the definition of iterators to be located within the container class. This has the advantage of establishing a strong link between the two implementations and of granting full access to the internals of the container object. Figure 1 shows a two dimensional matrix class which has a method, elements(), that returns an iterator object. The iterator extends the class SeqMutableIter of sequenceable and mutable iterators; it is an abstract class which defines a set of four standard methods: next() and prev() to return the next and previous elements of the sequence, set() and setNext() to modify the contents of the container. Note that it accesses the state of the matrix directly. Given a matrix r, the code to initialize all elements is:

```
r = m.elements();
loop
    r.setNext(null);
```

The efficiency of this solution depends on the ability of the compiler to in-line method calls and replace dynamic allocation with stack allocation. Recent work on static program analysis [1] and compiler optimizations of object-oriented programs [5] suggests that it is possible to generate good code for iterators.

An advantage of our approach is that many different iteration strategies can be added to a container class without placing a burden on the interface of the class. For instance, we can have an iterator to perform a row order traversal of the matrix quite

```
class Matrix {

 Object data [][];

 public Matrix (int x, int y) { data = new Object[x][y]; }

 SeqIter elements() {

    return new SeqMutableIter{
         int x = 0
         int y = -1;

         iter Object next() {
            if (x >= data.length)
                break;
            if (++y < data[0].length)
                return data[x][y];
            y = -1; x++;
            return next();
         }
         iter Object prev() {
            if (x < 0) break;
            if (--y >= 0)return data[x][y];
            y = data[0].length; x--;
            return prev();
         }
         boolean done() {
            return x<0 || x> data.length;
         }

         iter void set(Object o) {
            if (done()) break;
            return data[x][y];
         }

         iter void setNext(Object o) {
            next(); set(o);
         }
    }();
 }
 ...
```

Figure 1 A matrix class with an iterator.

easily. All that needs be done is to add the following method to the Matrix class:

```
Iter rows() {                              // rows() is a method of the Matrix class
    return new Iter{                       // newIter() returns an anonymous object
        int x = -1;                        // x is an instance variable of the iter
        iter Object next() {               // next() returns an anonymous object
            if (++x >= data.length)
                break;
            return MutableIter {           // The iterator traverses a row of the matrix
                int y = 0;                 // y indicates the position in the row
                iter Object next() {       // next() returns the next element
                    if (y >= data[0].length)
                        break;
                    else
                        return data[x][y++];
                }
            }();
        }
    } ();
}
```

The method next() of the outermost iterator returns a different element iterator each time it is invoked. The row iterator does not allow modification of elements, while the element iterator does. As an illustration, the following code fragment adds a different integer to each row:

```
r = m.rows();
loop {
    c = (MutableIter) r.next();
    val++;
    loop c.set( (Int) c.next().add(val));
}
```

Iterators can be composed. To illustrate how, we show a solution to the fringe comparison problem (i.e. compare the value of all leaves of two trees). A solution in Sather is given in [11]. In the following example we assume that the class tree has a method allElements which returns an iterator and a method isLeaf to check if a tree is a leaf.

```
class FringeTree extends Tree {
    public Iter fringe() {                         // fringe() returns an iterator
        Iter allElements = elements();             // allElements stores the standard tree iter
        return Iter{                               // The iterator returned by fringe() filters
            iter Object next() {                   // the values returned by elements() and
                loop {                             // yields only the leaves of the tree.
                    Tree t = (Tree) allElements.next();
                    if t.isLeaf() return t;
                }
                break;
            }
            boolean done() { allElements.done();}  // The method done returns true if all
        }();                                       // elements have been visited
    }
    boolean fringeCompare(FringeTree B) {          // fringeCompare() compares two trees.
        Iter iterA = fringe(), IterB = B.fringe();
        boolean check = true;
        while (check)
            check = (iterA.next()) .equals( iterB.next() ) ); // compare the fringes
        return check && iterA.done() && iterB.done();
    }
}
```

Translation Iterators are implemented using the exception mechanism of Java. Two new exceptions are defined, Break and Continue /*1*/. Inside an iterator method, a **break** at method level or labelled with the method name is translated to a **throw** of the exception Break /*2*/. Similar a **continue** is translated to a **throw** of the exception Continue. Each use of an iterator method is protected by a **try** which catches Break and Continue exceptions /*3*/. Only one **try** block for each loop nesting containing iterators is generated. The exception handler contains the **break** or **continue** statement optional extended by a label /*4*/. The **loop** keyword is just transformed into a **while** (true) /*5*/.

source

```
class Sequence {
   private int i, max;
   public iter int next() {
      if (i > max)
         break;                              /*2*/
      return i++;
   }
   public Sequence(int count) {
      max = count;
      i = 1;
   }
}
class itertest {
   public static void main(String args[]) {
      Sequence seq = new Sequence(10);
      loop
         System.out.println("Seq: "
            + seq.next());                   /*3*/
   }
}
```

translation

```
final class Break extends Exception {}       /*1*/
final class Continue extends Exception {}    /*1*/
class Sequence {
   private int i, max;
   public int next() throws Break, Continue {
      if (i > max)
         throw new Break();                  /*2*/
      return i++;
   }
   public Sequence(int count) {
      max = count;
      i = 1;
   }
}
class itertest {
   public static void main(String args[]) {
      Sequence seq = new Sequence(10);
      while (true)                           /*5*/
         try                                 /*3*/
            System.out.println("Seq: "
               +seq.next());
         catch (Break b) break;              /*4*/
         catch (Continue c) continue;
   }
}
```

Discussion The iterator concept presented here is more expressive than Sather's iters and allows an elegant encapsulation of iteration protocols. The translation scheme relies on closures and anonymous objects presented earlier. Exceptions are an efficient implementation technique. In Java they are very frequent and are often implemented as functions with two return values: the exception and the function return value. Our iterators are just that: methods with two return values. The exception set-up and the iterator call overhead can be eliminated if the iterator is inlined.

7 Conclusions

This paper proposes four non-intrusive language extensions, *tuples*, *closures*, *anonymous objects* and *iterators*, designed to improve the expressive power of Java without modifying the semantics of existing programs or complicating the language unduly. Tuples are useful on their own as typed heterogeneous containers of fixed size. Their real value is for efficiently implementing functions with multiple return values. In this respect they fit well with closures. Closures are first class functions. Their presence in the language makes it easy to express functions that manipulate other functions as is commonly done in functional programming. Anonymous objects are literal objects closely related to closures since they too can be nested in methods and classes. Their translation scheme follows the scheme described for closures. They are crucial for implementing iterators efficiently. Iterators are quite valuable as they permit encapsulation of iteration strategies within containers. In other words, the iteration algorithm, complete with its initialization code and termination condition, is hidden within the container data structure. This makes client code simpler and more robust. A translator implementing our extensions is available at http://www.complang.tuwien.ac.at/java/xjava/.

References

[1] O. Agassen, The Cartesian Product Algorithm, In Proc. ECOOP'95, 1995.

[2] A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools, Addison Wesley, 1986

[3] J.A. Banks, B. Liskov, A.C. Myers, Parameterized types and Java, MIT-LCS TM-553, May 1996.

[4] P. Bothner and R. Alexander Milowsk. The Kawa Scheme interpreter project. http://www.-winternet.com/~sgml/kawa, 1996.

[5] C. Chambers, D. Ungar and E. Lee, An efficient implementation of SELF, a dynamically type object-oriented language, Proc. OOPSLA'89, New Orlean, LA, 1989.

[6] M. Cowlishaw. NetRexx. http://www.ibm/com/Technology/NetRexx, 1996.

[7] J. C. Hardwick and J. Sipelstein. Java as an Intermediate Language. CMU-CS-96-161, Carnegie Mellon University, August 1996.

[8] P. Henderson, Functional Programming: application and implementation, Prentice-Hall, 1980.

[9] M. Lehman. HotTEA. http://www.mbay.net/~cereus7/HotTEA.html, 1996.

[10] B. Liskov, A. Snyder, R. Atkinson and C. Schaffert, Abstraction Mechanisms in CLU, CACM, 20(8), 1977.

[11] S. Murer, S. Omohundro, D. Stoutamire, C. Szyperski, Iteration Abstraction in Sather, TOPLAS 18(1) Jan 1996.

[12] J. Gosling, B. Joy and G. L. Steel, The Java Language Specification. Sun, 1996.

[13] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In Proc. POPL97, Paris, France, January 1997.

[14] Nick Shaylor, http://www.digiserve.com/nshaylor/jpp.html, 1996.

[15] Sun Microsystems, Inner classes in Java 1.1, Javasoft. http://java.sun.com/products/JDK/1.1/designspecs/innerclasses.

[16] Sun Microsystems, The Java Virtual Machine Specification, August 1995.

[17] S. Tucker Taft. Programming the Internet in Ada 95. Submitted to Ada Europe'96.