COEXISTENCE OF FUNCTIONAL AND OBJECT-ORIENTED PARADIGMS

Vijay Gehlot, Thomas Way, Frank Klassner
Department of Computing Sciences
Villanova University
800 Lancaster Avenue
Villanova, PA 19085-1699
610-519-7307
vijay.gehlot@villanova.edu, thomas.way@villanova.edu, frank.klassner@villanova.edu

ABSTRACT

"Objects-first" is a popular teaching approach in CS1/2 courses, particularly those designed around the Java programming language. Recently, theory-based criticisms of this approach have been used at some noteworthy universities to justify delaying object orientation to second year courses in favor of functional-based introductory curricula, and even treating object orientation as an optional advanced topic. In this paper we present a case for adopting a co-existential approach between object-orientation and functional programming based on the view that development of "computational thinking" skills should be a fundamental goal of CS1/2 courses. Towards this end, a tight relationship between ML and Java is explored and examined by means of several illustrative examples.

1. INTRODUCTION

The Final Report of the Joint ACM/IEEE-CS Task Force on Computing Curricula 2001 for Computer Science [1] identifies three implementations of a programming-first model, namely, imperative-first, objects-first, and functional-first. Depending on how the curriculum is structured, many such students will never get to see the benefits of other paradigms. Also, even when multiple paradigms are introduced, there is seldom an attempt to tie them together.

The foundational goal of Computer Science education is teaching students to be good computational thinkers [12]. Computational Thinking includes the use of abstraction, decomposition, and reformulation of problems to discover solutions, and these are critical skills we try to instill in our students, majors and non-majors alike. More broadly, the field of Computing itself increasingly is recognized as not simply an abstract way of expressing calculations, but as a natural science that describes fundamental processes by which much, if not all, of the world behaves [5]. Thus, a solid understanding of computing gained through development of Computational Thinking skills, is undeniably necessary. While there is general agreement that Computational Thinking as a broad concept will be a critical ability for the 21st Century, there is disagreement over how best to introduce students to this skill early in their Computer Science education during programming-first CS1 (or CS0) courses [2, 4].

The objects-first approach has gradually replaced the imperative-first approach as a student's first introduction to programming [4]. The choice of language for teaching programming aligns with the nature of programming in general, which has seen popular programming languages used in academic and industrial settings evolve from imperative languages Fortran, Pascal, and C to object-oriented C++, Java, and C# [7].

Functional programming languages such as Lisp and ML, on the other hand, have never experienced the wide adoption of C or Java, for instance, but neither has their popularity diminished for specific programming needs and for teaching.

At Carnegie Mellon University (CMU), a serious study of how Computer Science is taught was conducted, with the conclusion being that the objects-first approach introduces unnecessary complexity and can discourage students who are new to programming. CMU has concluded that restructuring the order in which students are introduced to programming and the ideas of Computational Thinking is best

served by initially using an imperative-first or functional-first approach and introducing students to object-oriented concepts later [2].

Clearly, this is an important development in Computer Science education, although it is less likely an indication that the objects-first approach has failed [3], and more likely an indication that introductory students benefit from a more basic approach to learning how to program in an imperative style while still using Java as a language [9]. While most students will receive an excellent grounding in programming and Computational Thinking regardless of the language employed, the counter-trend being proposed at CMU appears to be towards a diverse and basics-first approach that introduces students to programming without overemphasis of object-orientation. Students can benefit from exposure to multiple languages, and through this exposure gain the deeper understanding that come through making the connections between a variety of computational approaches and languages.

We illustrate our approach toward applying a basics-first by providing several illustrative examples that allow students to apply the basics to ML [10] and to Java [8] in very clear ways. In fact, the approach we present can be fully automated. Since objects are (compound) data together with desired behavior, we begin by giving a language-independent construction of compound data in the next section. In Section 3 we relate ML and Java for each such construction by showing that if syntactic details are ignored then both languages support the same construction. The differences arise only in terms of how the data is manipulated. Our conclusions are presented in Section 4.

2. COMPOUND TYPES

A compound type is a type whose values are constructed from other types. Although different programming languages support differing mechanisms for forming compounds, they all can be understood in terms of some simple operations on sets if we view a datatype as a set of values together with some suitable operations [11]. For the purposes of this paper, we consider the following three set construction operations (we omit discussion of fourth construction, namely, arrow or function type here):

- **Cartesian product**: Given two sets *S* and *T* we have:

  $$S \times T = \{(x, y) | x \in S, y \in T\}.$$

  This construction generalizes to more than two sets. Note that empty product is a well defined type according to this definition and contains exactly one element. The `void` type of Java and `unit` type of ML denote the empty product.
- **Disjoint union**: Given two sets *S* and *T* we have:

  $$S \uplus T = \{inl\ x\ |x\ \in S\} \cup \{inr\ y\ |y \in T\}.$$

  Here values chosen from *S* have been tagged *inl*, and values chosen from *T* have been tagged *inr*. This construction generalizes to more than two sets. Furthermore, if each of the types is an empty product we get precisely an enumeration type. Thus, one does not need, in principle, a separate construction for enumeration types. Using this fact and the representation of union types in Java shown in Section 3.5 below, one can create type safe enum in pre-1.5 versions of Java. On the other hand we can understand the type-safe enums of Java 1.5 [6] as a special case of union types.
- **Recursion**: A recursive type *S* is defined by a set equation of the form:

  $$S = \ldots S \ldots$$

  In general, there may be many such mutually recursive set equations. A well defined recursive equation will normally involve product and union operations. Thus, if we can understand products and unions, we can easily and directly code recursive types. We'll use this fact to create a very clean implementation of expression trees in Java in Section 3.9 below.

Next, we show how each of these constructions are supported in ML and Java. In fact, for Java we first construct a compound type without resorting to the object view thereby clearly separating the construction of a compound data from the operations associated with it and then we reorganize the two pieces to give us the object view.

3. RELATING ML AND JAVA

In this section we consider an example of each type of construction in ML and show its translation to Java. Note that the non-object view presented below means that the functions that operate on the associated data (object) are not part of the class definition itself. Specifically, the use of the static method declarations eliminates instantiation of objects, enabling Java effectively to be used in an imperative style.

3.1 Products in ML

Products can be defined in terms of either tuples or records in ML. Here is an example of such a construction:

```
(* declaration of type *)
type PERSON = { NAME: string,  AGE : int }
(* a useful operation on this type *)
fun getAge(p:PERSON):int = #AGE(p)
(* creation of a value of this type *)
val p1:PERSON = {NAME="Joe", AGE=12}
```

3.2 Products in Java—Non-objects View

We can use Java's class facility to create a product type. The operations on such a type will reside outside of the type definition under the non-objects view. Here is a Java translation of the above piece of ML code:

```
// declaration of type
public class Person {
    String name;
    int age;
    Person(String s, int a) {
        name = s;
        age = a;
    }
}
```

```
public class UsefulFunctions {
    static int getAge(Person p) {
        return p.age;
    }
...
//other functions for this type
}
...
// value creation of this type
Person p1 = new Person("Joe",12);
```

3.3 Products in Java—Objects View

If we view a person as an object then the functions associated with these objects become part of the associated class declaration. This is a mere reorganization of pieces presented above in Section 3.2. With this reorganization, the function `getAge` no longer needs a parameter as shown below:

```
public class Person {
    String name;
    int age;
    Person(String s, int a) {
        name = s; age = a;
    }
    int getAge() { return age; }
}
```

Although simple, this example demonstrates a clear distinction between the objects view versus the non-objects view and helps us see that the data and associated operations are separate concerns. Ignoring syntactic details, we can also see the tight similarity between the ML code and the corresponding Java code.

3.4 Unions in ML

ML has the `datatype` declaration that directly corresponds to the disjoint union construction described above. Here is an example:

```
(* A union type representing two shapes *)
datatype shape =
    Rectangle of real * real (* the two sides *)
  | Circle of real; (* the radius *)
(* a useful function over this type *)
fun area (Rectangle(x,y)) = x * y
  | area (Circle r) = Math.pi * r * r;
```

3.5 Unions in Java—Non-objects view

Java does not have direct operators to create a union type; however, we can use other facilities in Java to achieve this construction. Towards this end, consider the declaration of type `shape` above in ML. The type being defined on the left hand side represents an abstract entity whereas the right hand side represents the concrete pieces that make up this entity. This is precisely the relationship between an abstract class and its (concrete) subclasses. Thus we can code this union type in Java as follows:

```
public abstract class Shape { } // the abstract entity
public class Rectangle extends Shape{// concrete subclass
    public final double width;
    public final double height;
    public Rectangle(double w, double h) {
        width = w; height = h;
    }
}
public class Circle extends Shape {// concrete subclass
    public final double radius;
    public Circle(double r) {
        radius = r;
    }
}
// Collect together useful functions over shapes in a utility class.
public class UsefulFunctionsOnShapes {
    public static double area(Shape s) {
        if (s instanceof Rectangle) {
            return ((Rectangle) s).width *((Rectangle) s).height;
        } else {// it is a Circle
            return Math.PI * ((Circle) s).radius * ((Circle) s).radius;
        }
    }
    // ... other useful functions on shapes
}
```

3.6 Unions in Java—Objects view

Again, taking the objects view merely involves reorganization of functions as discussed above. Also, the reorganization makes it clear that any common functionality of the concrete classes can, and should, be included in the objects view of the abstract class. Thus, we arrive at the following:

```
public abstract class Shape {// the abstract entity
    // specification of common functionality
    public abstract double area();
}
public class Rectangle extends Shape{// concrete subclass
    public final double width;
    public final double height;
    public Rectangle(double w, double h) {
        width = w; height = h;
    }
    // concrete definition of abstract functionality
    public double area() {
        return width * height;
    }
}
```

```
public class Circle extends Shape {// concrete subclass
    public final double radius;
    public Circle(double r) {
        radius = r;
    }
   // concrete definition of abstract functionality
    public double area() {
        return Math.PI * radius * radius;
    }
}
```

3.7 Recursive Types in ML

Consider the following definition of a type to create expression trees consisting of integers and the binary operations of summation and multiplication:

$$Expr = Integer \uplus (Expr \times Expr) \uplus (Expr \times Expr).$$

Since we have already seen the representation of products and union types, we can directly code this type in ML as follows:

```
datatype Expr =  Num of int  | Plus of Expr * Expr  | Times of Expr * Expr
```

Note that `Num`, `Plus`, and `Times` are merely tags. One useful operation on such expressions is that of evaluation. Using the pattern matching facility of ML we can define this operation fairly compactly as follows:

```
fun eval(Num v)          = v
  | eval(Plus(e1, e2))   = eval(e1) + eval(e2)
  | eval(Times(e1, e2))  = eval(e1) * eval(e2)
```

3.8 Recursive Types in Java—Non-objects view

Using our approach to translating product and union types in Java described in previous sections, we arrive at the following code:

```
public abstract class Expr {}
public class Num extends Expr {
    public final int value;
    public Num(int i) {
        value = i;
    }
}
public class Plus extends Expr {
    public final Expr exp1, expr2;
    public Plus(Expr e1, Expr e2) {
        exp1 = e1; exp2 = e2;
    }
}
public class Times extends Expr {
    public final Expr exp1, expr2;
    public Times(Expr e1, Expr e2) {
        exp1 = e1; exp2 = e2;
    }
}
```

```
public class UtilityClass {
// the eval function
   public static int eval(Expr e) {
        if (e instanceof Num) {
            return ((Num)e).value;
        } else if (e instanceof Plus) {
            return eval(((Plus)e).exp1) +
                   eval(((Plus)e).exp2);
        } else {
            return eval(((Times)e).exp1)*
                   eval(((Times)e).exp2);
        }
    }
}
```

3.9 Recursive Types in Java—Objects view

The reorganization of the code above gives us the following objects view of expression trees:

```
public abstract class Expr {
    abstract public int eval();
}
class Num extends Expr {
    public final int value;
    public Num(int i) {
        value = i;
    }
    public int eval() {
        return value;
    }
}
public class Plus extends Expr {
    public final Expr exp1, expr2;
    public Plus(Expr e1, Expr e2) {
        exp1 = e1; exp2 = e2;
    }
    public int eval() {
        return exp1.eval() + exp2.eval();
    }
}
public class Times extends Expr {
    public final Expr exp1, expr2;
    public Times(Expr e1, Expr e2) {
        exp1 = e1; exp2 = e2;
    }
    public int eval() {
        return exp1.eval() * exp2.eval();

    }
}
```

In the example above, the code produced is clean and semantically elegant. In fact, a good computational thinker should be able to recognize the pattern and automate the task. From a computational thinking perspective, the approach outlined above helps students think in terms of *what to code* as opposed to the *how to code* approach that typically gets emphasized in a single paradigm setting.

4. CONCLUSION

There is a concern that the objects-first approach adopted by many institutions may be developing into a *de facto* objects-only approach. We argued the benefits of a multi-paradigm approach to programming. We presented a language independent definition of compound types and then showed how to systematically code these in two different languages. Another benefit of this approach is to exhibit a tight connection between ML and Java. This approach is suitable to connecting other languages too. For example, Java and Scheme, or C and ML, or Scheme and ML. We plan to incorporate successful aspects of CMU's new approach into our future curriculum revisions.

REFERENCES

[1] ACM/IEEE-CS, Final report of the joint acm/ieee-cs task force on computing curricula 2001 for computer science, 2001, www.computer.org/education/cc2001/final/index.htm, retrieved September 10, 2004.

[2] Anderson, D., Blelloch, G., Harper, R., Lafferty, J., Pfenning, F., Platzer, A., Sleator, D., Stehlik, M., Recommendations for revising introductory computer science, Carnegie Mellon University, Internal report, February 2010.

[3] Astrachan, O., Bruce, K., Koffman, E., Kölling, M., Reges, S., Resolved: objects early has failed, *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '05)*, 451-452, March 2005.

[4] Cooper, S., Dann, W., Pauch, R., Teaching objects-first in introductory computer science. *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '03)*, 191-195, March 2003.

[5] Denning, P. J., Computing is a natural science. *Communications of the ACM*, 50(7):13-18, July 2007.

[6] Frens, J. D., Taming the tiger: teaching the next version of Java, *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '04)*, 151–155, March 2004.

[7] Goosen, L., A brief history of choosing first programming languages, Proceedings *International Federation for Information Processing (IFIP)*, Volume 269, History of Computing and Education 3, John Impagliazzo (ed.), Springer, 167-170, 2008.

[8] Lewis, J., Loftus, W., *Java Software Solutions—Foundations of Program Design*, Addison Wesley, 6th edition, 2009.

[9] Reges, S., Back to basics in CS1 and CS2, *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '06)*, 293-297, March 2006.

[10] Ullman, J. D., *Elements of ML Programming*, Prentice Hall, ML97 edition, 1998.

[11] Watt, D. A., *Programming Language Concepts and Paradigms*, Prentice Hall, 1990.

[12] Wing, J. M., Computational thinking, *Communications of the ACM*, 49(3):33-35, March 2006.