Elements of a Reversible Object-Oriented Language
Work-in-Progress Report

Ulrik Pagh Schultz[1(✉)] and Holger Bock Axelsen[2]

[1] University of Southern Denmark, Odense, Denmark
ups@mmmi.sdu.dk
[2] University of Copenhagen, Copenhagen, Denmark
funkstar@di.ku.dk

Abstract. This paper presents initial ideas for the design and implementation of a reversible object-oriented language based on extending Janus with object-oriented concepts such as classes that encapsulate behavior and state, inheritance, virtual dispatching, as well as constructors. We show that virtual dispatching is a reversible decision mechanism easily translatable to a standard reversible programming model such as Janus, and we argue that reversible management of state can be accomplished using reversible constructors. The language is implemented in terms of translation to standard Janus programs.

1 Introduction

Extant reversible programming languages such as Janus [7], Theseus [3] and RFUN [8] have been developed with a focus on providing features (such as control flow operators) that enables the programmer to understand how execution is performed reversibly. However, unlike most modern programming languages, this is usually *not* paired with other programmer-friendly abstractions. This has unfortunate consequences, in particular that reversible programmers have to build implicit data types out of the given primitives when dealing with complex data, leading to longer, less readable, and more error-prone reversible code.

From recent advances in compiler technology for reversible programming languages we know that it is possible to reversibly and efficiently represent and manipulate complex data objects in the heap [2,6], opening the door for associated advances in reversible language design. Here, we consider *reversible object-orientation*. Object-oriented programming uses classes as a means to providing higher-level structures that encapsulate behavior and state. We show how a number of object-oriented concepts (encapsulation, inheritance, and virtual methods) can be captured reversibly by extending the Janus language with support for

The authors acknowledge partial support from COST Action IC1405 *Reversible Computation*. H.B. Axelsen was supported by the Danish Council for Independent Research | Natural Sciences under the *Foundations of Reversible Computing project.*

S. Devitt and I. Lanese (Eds.): RC 2016, LNCS 9720, pp. 153–159, 2016.
DOI: 10.1007/978-3-319-40578-0_10

such features, and describe them by translation to ordinary Janus programs. These concepts have been implemented in a prototype language named Joule (a homonym of JOOL, Janus Object-Oriented Language), which will be used throughout this paper to illustrate our ideas. This paper presents initial concepts in the design of the Joule language, serving as a report on the work in progress to provide a useful, reversible object-oriented programming language.

2 Reversible Object-Oriented Programming

Similarly to mainstream object-oriented languages such as C++ and Java, we propose to extend Janus with a static inheritance mechanism encapsulating state and behavior, and a corresponding virtual dispatching mechanism that dynamically decides which method implementation to invoke based on the runtime type of the receiver object. We hypothesize that such a language will allow programs to be written at a higher level of abstraction without introducing complications due to memory management.

Object-oriented polymorphism is implemented using inheritance, where operations are expressed in terms of an abstract interface implemented by subclasses. Polymorphism allows different implementations to be composed and then selected at runtime depending on the specific class of each object. Since objects do not change their class at runtime in our proposed language, the decision of which method to invoke at runtime will be reversible: invoking and "uninvoking" a specific method on a given object will always select the same method.

Regarding memory management, some object-oriented languages have been conceived with limited-memory systems in mind, and today they are routinely used to implement embedded systems. For example, the Beta language (an early derivative of Simula-67, the first object-oriented language) included static object allocation as a design criterion [4], to enable it to function on memory-constrained systems with static and stack allocation. Today, the C++ language is commonly used as a systems programming language: the combination of object-oriented system decomposition and a disciplined approach to manual memory management often offers significant advantages compared to, e.g., C.

3 Encapsulation and Construction

We now describe how classes are used as an encapsulation mechanism in Joule, our proposed syntax for reversible method invocation, and how we propose to deal with the issue of reversibly constructing objects.

3.1 Encapsulation

Object-oriented classes should not be considered as a module mechanism, but classes have nonetheless been proven as a practical mechanism for providing the encapsulation and abstraction required for, e.g., abstract datatypes [5]. Taking

```
class Point {
  int x; int y; // private fields, zero-initialized
  Point(int x, int y) { // constructor, runs after allocation
    this.x += x; this.y += y; // 'this.x' is a field, 'x' a parameter
  }
  procedure add_to_x(int x) { this.x += x; }
  procedure add_to_y(int y) { this.y += y; }
}
```

Fig. 1. Joule implementation of a basic point class

inspiration from mainstream languages, we can allow classes to define fields and methods that can operate on the data stored in these fields. The data is initialized using a constructor and uninitialized by uncalling the constructor.

As a concrete example, we define a class `Point` that encapsulates two values, x and y coordinates, and provides operations to manipulate these values (see Fig. 1). The fields `x` and `y` can only be manipulated using the provided methods (we consider all fields private). The fields are initialized upon object initialization using the constructor. Note that the initial value of any field is assumed to be zero (or null for a reference type). Joule objects can be considered as records that contain a mix of runtime type information, integers, and object references (the specific Janus-based implementation will be discussed later, in Sect. 5).

The class `Point` can be instantiated and methods can be invoked (called) using the standard "." operator for accessing an object. To support uncalling method (uninvoking), we adopt "!" as an inverse operator.

```
local Point p = Point.new(5,8); // construct
p.add_to_x(2); // p.x==7
p!add_to_y(3); // p.y==5
```

Note the slightly nonstandard syntax `C.new(...)` for creating an object and invoking the constructor, which in Java would have been written `new C(...)`. Calling and uncalling methods works similar to calling and uncalling procedures in Janus. Nevertheless, the introduction of a class hierarchy will require a runtime decision to select which implementation to use, as discussed in the next section.

3.2 Construction and Unconstruction

To properly dispose of a locally allocated object we must restore the value of the fields to their initial blank values from before the constructor was invoked.[1] To

[1] We here follow the memory model of Janus, where variables can be dynamically allocated on the call stack using a `local` declaration that initializes the variable to a given value, but must symmetrically by deallocated using a `delocal` declaration that must provide the final value of the variable, resetting the memory and providing an initializer for the variable when running the program in reverse.

this end, we propose to uncall the constructor using arguments that return the corresponding fields to zero (or null for references). The locally allocated variable `p` of type `Point` now representing the point $(7, 5)$ can for example be disposed using `delocal Point!new(7,5) p;` The "!" operator is used here to denote running the constructor in reverse with the given arguments, unconstructing the object.

```
class Counter {
  int limit; // stop incrementing this.count when limit is reached
  int value; // updated when calling 'count'
  Counter(int limit) { this.limit += limit; }
  procedure count(int flag) {
    if(this.value<this.limit) { this.value += 1; }
    else { flag += 1; } fi(flag==0);
  }
  procedure finalize(int uncount) { this.value -= uncount; }
}
```

Fig. 2. Joule implementation of counting up to a limit

In general objects may contain state that evolves over time and that is not initialized using constructor parameters. As an example, consider the class `Counter` shown in Fig. 2. The field `limit` is initialized upon construction, but the field `value` evolves over time: as long as its value is less than `limit` it is increased by one when the method `value` is called (the parameter `flag` is used to signal when the limit has been reached). Uncalling the constructor would not serve to return the field `value` to a zero state. Here we could adopt the notion of a destructor to reset the remaining state, but as an alternative we adopt a simple programming pattern where a method (by convention named `finalize`) is used to bring the object back to a state where it can be unconstructed by running the constructor in reverse:

```
local Counter c = Counter.new(3); // construct
local int flag = 0;
c.count(flag); c.count(flag);
c.finalize(2);                    // reset c.value to 0
delocal flag == 0;
delocal Counter!new(3) c;         // unconstruct
```

The method `finalize` serves to "unfinalize" the object, bringing it into a state where the constructor can be run backwards to reset the memory. In the concrete example the finalization method takes an argument, but the finalization could also have written with the assumption that the counter is in a specific state (e.g., limit reached), in which case no argument would have been needed.

We speculate that the question of how to unconstruct objects will be a key challenge in reversible object-oriented programming, but that a notion of *reversible design patterns* may provide useful programming abstractions. For example, objects created by a factory design pattern could then be unconstructed by a hypothetical *unfactory* pattern derived from the original factory pattern. This issue is however left for future work.

3.3 Object References

Most object-oriented programs rely on the ability for objects to refer to each other, which raises the question of how to reversibly store references to other objects in a field. We adopt the simple approach that references only can be stored into null references, which is done using the `:=` operator:

```
local Point p = Point.new(1,7); local Point q = null;
q := p;                   // essentially q += p;
q.add_to_x(2);            // p.x==3
delocal q == p;           // removes local variable
delocal Point!new(3,7) p; // unconstructs object
```

Reverse execution of the `:=` operator is done by subtracting the provided reference from the reference being operated on, producing a null reference.

4 Inheritance and Virtual Calls

Inheritance often serves the dual purpose of creating a subtype hierarchy and implementation reuse, and for simplicity we follow this approach here. Although concepts of reverse inheritance have been proposed [1], we see inheritance and the subtype hierarchy as a means to model the information on which the methods operate. Thus, we believe that inheritance works the same in non-reversible and reversible languages, although as noted earlier the immutability of type information in an object is particularly advantageous for reversible computing since it ensures that virtual calls are a reversible mechanism.

Our proposed syntax for calling and uncalling methods has already been introduced, and straightforwardly generalizes to invocation of virtual methods. As a concrete example, consider the Joule program shown in Fig. 3. The classes `Add`, `Sub` and `Twice` all extend the common (abstract) superclass `Op`. The class `Twice` takes a given operator as an argument, and applies it twice whenever the `app` method is called (note the use of the reversible `:=` null-reference assignment operator). These classes can be used as follows:

```
abstract class Op { abstract procedure app(int var, int x); }
class Add extends Op { procedure app(int var, int x) { var += x; } }
class Sub extends Op { procedure app(int var, int x) { var -= x; } }
class Twice extends Op {
  Op p;
  Twice(Op p) { this.p := p; } // := only on null references
  procedure app(int var, int x) {
    local Op p = this.p; // copy of reference, fewer indirections
    p.app(var,x); p.app(var,x); // Polymorphic call site
    delocal p == this.p;
  }
}
```

Fig. 3. Joule program implementing a hierarchy of reversible operators

```
local Op a = Add.new(); local Op b = Sub.new();
local Op aa = Twice.new(a); local Op bb = Twice.new(b);
local int v = 0; aa.app(v,4); bb!app(v,1); delocal v == 10;
delocal Twice!new(b) bb; delocal Twice!new(a) aa;
delocal Sub!new() b; delocal Add!new() a;
```

Here, the calls `p.app` inside the method `app` in class `Twice` execute different methods, depending on the type of the `Op` object stored in the field `p`.

5 Implementation

Joule has been implemented by translation to Janus, and all the examples provided in this paper have been automatically compiled using our prototype Joule compiler.[2] Objects are currently represented as arrays of integers: the first element is a compile-time constant determining the class of the object, the remaining elements represent the fields of the object. Objects are allocated in a heap represented as a two-dimensional array, thus object references are simply indices into this array. The dimensions of the array are currently determined manually, and memory management is currently manual and completely unsafe, meaning objects could be deallocated in the wrong order leading to undefined behavior (e.g., an object could be overwritten by user data).

Virtual calls are implemented using standard dispatcher functions, e.g., for a given virtual method `m` a Janus procedure `dispatch_m` is generated that uses nested if-then-else-fi statements to select the specific method implementation to call depending on the type of the receiver object (the value of the first element of the array representing the receiver object). Uncalling a method is simply done by uncalling the corresponding dispatcher procedure. This implementation approach is simple and supports compile-time modularity (e.g., classes can be written independently) but rules out runtime modularity (e.g., dynamic class loading). Since reversible computing normally operates under a closed-world hypothesis, this restriction is considered appropriate for the time being.

References

1. Chirila, C.B., Crescenzo, P., Lahire, P.: Reverse inheritance: improving class library reuse in Eiffel. In: Langages et Modeles a Objets (2007)
2. Hansen, J.S.K.: Translation of a reversible functional programming language. Master's thesis, Department of Computer Science, University of Copenhagen (2014)
3. James, R.P., Sabry, A.: Theseus: a high level language for reversible computing, work-in-progress report at RC (2014). http://www.cs.indiana.edu/~sabry/papers/theseus.pdf
4. Kristensen, B.B., Madsen, O.L., Møller-Pedersen, B.: The when, why and why not of the beta programming language. In: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, pp. 10-1–10-57. HOPL III, NY, USA (2007). http://doi.acm.org/10.1145/1238844.1238854

[2] Source code for compiler, examples, and generated Janus programs are available at https://github.com/joule-lang/joule/tree/master/doc/papers/rc16.

5. Meyer, B.: Object-Oriented Software Construction, vol. 2. Prentice Hall, New York (1988)
6. Mogensen, T.: Garbage collection for reversible functional languages. In: Krivine, J., Stefani, J.B. (eds.) RC 2015. LNCS, vol. 9138, pp. 79–94. Springer, Heidelberg (2015)
7. Yokoyama, T., Axelsen, H.B., Glück, R.: Principles of a reversible programming language. In: Proceedings of Computing Frontiers, pp. 43–54. ACM (2008)
8. Yokoyama, T., Axelsen, H.B., Glück, R.: Towards a reversible functional language. In: De Vos, A., Wille, R. (eds.) RC 2011. LNCS, vol. 7165, pp. 14–29. Springer, Heidelberg (2012)