UDK 519.681.2

Building Cyclic Data in a Functional-Like Language Extended with Monotonic Objects

Alexei I. Adamovich (Ailamazyan Program Systems Institute of RAS),
Andrei V. Klimov (Keldysh Institute of Applied Mathematics of RAS)

We discuss an approach to deterministic parallel programming based on a two-level programming language. It comprises a higher-level functional-like subset for application programmers and a lower-level object-oriented Java-like language for experts in parallel programming, who develop libraries of classes. The experts guarantee determinism for the higher-level language user. We refer to these classes and objects as *monotonic* and give their definition as preserving two properties of the higher-level programs: determinism and idempotency. In this case study, we address the problem of representing cyclic data structures, which is unsolvable in purely functional languages, easy in object-oriented languages, and solvable but tricky with monotonic objects. As an introductory example, a simple monotonic class is given—a variable of a primitive type. Then we show that an analogous class declaration for a variable of a reference type is non-monotonic and reveal that building cyclic data structures in this setting is nontrivial. Finally, we present examples that demonstrate some of the subtleties of designing monotonic classes. This paper describes a work-in-progress towards a theory and a software system for deterministic parallel programming. It also poses new problems for program verification to assist in proving that class declarations are monotonic and, therefore, parallel programs developed in the proposed system are proved deterministic.

Keywords: *deterministic parallel programming, functional languages, object-oriented languages, monotonic objects and classes.*

1. Introduction and Related Work

Parallel/concurrent programming and debugging are complicated because parallel/concurrent programs are nondeterministic in the general case. To simplify programming, various specific model of parallelism and concurrency have been, and are continuously being, invented. Some of them focus on the determinism of the results of computation, where all runs result with equivalent final states despite of interleaving and different ordering of operators from different parallel/concurrent threads.

Accepted to X Workshop “Program Semantics, Specification and Verification: Theory and Applications”
(PSSV-2019, July 1 and 2, 2019, Novosibirsk Akademgorodok, Russia)
http://persons.iis.nsk.su/en/pssv2019

Please refer to our recent paper [5] for a survey of an extensive field of research on deterministic parallel/concurrent programming. Let us list below just some of these works which are the most interesting for our study.

The most restricted models are those based on pure functional programming, where side effects are absent, threads are independent of each other, and hence the results are always the same. Not surprisingly, active research on deterministic parallelism is carried out in the community of the purely functional language Haskell [10]. Our work is in fact a transfer of the ideas of the functional paradigm to object-oriented languages, while retaining some important properties in a more general form.

In the object-oriented setting, the work on Deterministic Parallel Java (DPJ) [6,7,11] extends the Java language and the compiler by adding certain features that guarantee determinism using an analysis that checks that parallel threads interfere only in a disciplined way, which does not violate determinism.

The most interesting from our point of view are results by Lindsey Kuper *et al*. [12,13,14,15] where she suggests that variables shared between threads should change monotonically in some partially ordered set (more precisely, a semilattice) and operations on them are defined in such a way that the result of computation is deterministic. In our work, we use this idea in a more general object-oriented setting.

The rest of the paper is organized as follows. In Section 2 we introduce the idea of a two-level programming language and system. In Section 3 the notion of a monotonic class and object is defined. In Section 4 an introductory example of a monotonic class is given. Sections 2–4 are based on our previous work [4,8,9]. In Section 5 we present novel material: using some examples we discuss problems and solutions on how to build cycle data structures using monotonic objects, thereby overcoming the limitations of purely functional programming, while preserving the determinism of parallel computation.

2. A Two-Level Programming Language and System

In order to meet controversial requirements of program determinism for application programmers and diversity of deterministic parallel computation models, we suggest the use of (and develop) a two-level programming language and system:

- The higher-level language is like a functional subset of Java with simple means to initiate new threads on function calls and method invocations (often called “futures”, “promises”[1]).

[1] https://en.wikipedia.org/wiki/Futures_and_promises

Threads create objects of classes declared at the lower level and communicate only through operations on them. Any syntactically correct program here is deterministic by construction. This higher-level language is intended for use by the application developers.

- The lower-level language is full Java, or any other similar object-oriented language, which comprises a complete set of means for concurrent programming of generally nondeterministic programs. In this language, experts in parallel and concurrent programming declare lower-level classes in such a way that their use in the higher-level language guarantees determinism and preserves other valuable properties discussed below. We refer to these classes and their objects as *monotonic*.

Notice a subtlety of this definition: the declarations of monotonic classes belong to the lower level, while their monotonicity is defined through the properties of programs in the higher-level language, rather than the properties of the classes *per se* like pre- and post-conditions of their methods, object invariants, *etc*.

The development of this language and system is based on (and continues) our previous research into parallel programming systems [1,2,3,4].

3. The Notion of a Monotonic Class and Object

To formally articulate what determinism means we need the notion of equivalence of computations. We use the Leibniz notion of contextual equivalence.

Definition 1 (equivalence of values). Two values are (*contextually*) *equivalent* if they are indistinguishable programmatically in the higher-level language, that is:

- the values are of the same type, and
- in the case of a primitive type: the values are equal, and
- in the case of a reference type: any function or method with a primitive result type, when applied to these values, returns equal results, or neither terminate. □

Definition 2 (equivalence of computations). Two executions of copies of an expression (with the copies of arguments) are (*contextually*) *equivalent*, if

- they both terminate and return equivalent values, or
- they both throw exceptions (which may be different), or
- neither terminate. □

Definition 3 (monotonic). A declaration of a class and the objects of the class are called *monotonic* if any program in the higher-level language using the operation of object creation and methods on the objects of this class, satisfies the following properties:

- *Determinism* of computation (or *confluence*), that is, the results obtained in different order of a parallel/concurrent computation are equivalent.
- *Idempotency*, that is, a repeated computation of a copy of an expression is equivalent to the original computation, and the difference between the side effects of the two runs is not observable programmatically in the higher-level language.

These properties must be satisfied simultaneously for all monotonic classes when they are used together in any program in the higher-level (functional-like) language. □

One may wonder about motivation behind the idempotency property. It may seem that idempotency is implied by determinism. However, in the definition of determinism, to compare the results of different computation order, repeated runs are performed starting from the same initial state, while in the definition of idempotency, the next run uses the final state of the previous one.

```
public class IntVar {

  boolean defined = false;
  int value;

  public synchronized int get()
  {
    if (!defined) wait();
    return value;
  }

  public synchronized void set(int x)
  {
    if (!defined) {
      value = x;
      defined = true;
      notifyAll();
    }
    else if (value != x)
      throw new RuntimeException();
  }
}
```

Fig. 1. Monotonic class `IntVar` with one field of the primitive type `int`.

```
public class ObjectVar {

  boolean defined = false;
  Object value;

  public synchronized Object get()
  {
    if (!defined) wait();
    return value;
  }

  public synchronized void set(Object x)
  {
    if (!defined) {
      value = x;
      defined = true;
      notifyAll();
    }
    else if (value != x)
      throw new RuntimeException();
  }
}
```

Fig. 2. Non-monotonic class `ObjectVar` with one field of the reference type `Object`.

4. A Simple Monotonic Class Example

Parallel threads communicate by means of side effects on variables of various types. Our goal is to construct monotonic versions of classes representing such variables.

Consider the case where a variable is of primitive type `int`. The monotonic class `IntVar` shown in Fig. 1 declares 2 methods `set` and `get` with the following semantics:

- `set(x)` stores the value `x`, if an unequal value has not been stored already; otherwise throws an exception;

- `get()` returns the value stored by `set()`, or waits until `set` has been invoked, and then completes.

Thus, each `IntVar` object monotonically changes from the undefined state to the state defined with some integer and then possibly to raising an exception, which may be read as the "overdefined" state. We argue that such behavior satisfies the definition of monotonic objects. This is explained in more detail in our earlier paper [9].

5. Building Cyclic Data Structures

Notice that it is principally impossible to build a cyclic data structure without using mutable data. That is why purely functional languages allow us to efficiently manipulate only trees. This prohibits development of high-performance software that manipulates graphs, which imperative and object-oriented languages allow. In order to efficiently manipulate graphs, where cyclic relations between vertices and edges are denoted by references, one needs to mutate the representation. Our goal is to allow mutable data *and* preserve the main properties of functional languages, which we capture in the notion of monotonic objects.

Let us study various examples of declarations of mutating operations on objects and see which of them are monotonic and which are not.

Example 1: non-monotonic. Let us change in Fig. 1 the `value` field from the primitive to reference type. Figure 2 shows the code of the class `ObjectVar`, which coincides with the monotonic class `IntVar`, except for the type `Object` instead of `int`. However, this makes the class non-monotonic. The cause of this is the lack of *referential transparency* of the `new` operator as it generates a new reference to a new object on each evaluation, which fundamentally differs from the world of functional programming. Consider the following code fragment:

```
ObjectVar a = new ObjectVar();
a.set(new ObjectVar());    // first evaluation of an expression
a.set(new ObjectVar());    // second evaluation of the same expression
return a.get();
```

The notion of monotonicity requires that reevaluation of an expression returns an equivalent result and is idempotent with respect to side effects (that is, nothing changes). However, in the second invocation of the method `set`, the condition `(value != x)` in its body evaluates to `true` and the exception `RuntimeException` is thrown.

Example 2: non-monotonic. A natural idea to avoid this unpleasant exception is to replace the comparison of references with the comparison of the objects' contents by the method `equal`:

```
if (!value.equal(x)) throw new RuntimeException();
```

The method `equal` should perform deep comparison, paying no attention to the equality of references, except for the sake of optimization, which is invisible from the outside, and in order to avoid looping when traversing cyclic data.

Nevertheless, the class `ObjectVar`, thus defined, is still non-monotonic, because the result of `a.get()` depends on the order of evaluation of the two `new` expressions. We imply that the statements `a.set(new ObjectVar())` can be computed in parallel, hence the result can be either the reference to the object created by the first `new` sub-expression, or by the second one. This difference is programmatically visible.

Example 3: monotonic. To fix this, we must avoid returning from monotonic objects (by methods like `get`) the references passed in arguments of any of its methods (like `set`). This can be done in two ways. First, a clone of the stored object can be created in `set`, the reference to which is then returned by all invocations of `get`. Second, a new clone of the stored object can be generated on each invocation of `get`. The first version seems more efficient (in terms of the memory for extra objects generated in the second version). However, either of these versions could be useful, depending on the application. A library of monotonic classes should contain both versions, with different names.

However, this does not complete the definition of the monotonic `ObjectVar` semantics. The objects in the argument of the `set` method could be undefined, or partially defined. The latter case may occur when the object has many fields, and some of them are already defined, while others are not. Thus, a kind of unification of the objects from the arguments of several invocations of `set` is required, the result of the unification being stored in the `ObjectVar` object. Now, if properly formalized and coded, the class `ObjectVar` becomes monotonic.

Nevertheless, some degrees of freedom preserving monotonicity remain. Should the unified objects be changed as well? Should the information about the unified objects flow in one direction from the `set` arguments to the stored object only, or could it flow in the opposite direction as well? The answer is that both versions are monotonic, and their usefulness depends on the application.

Example 4: building a cycle. Now we can build a cycle of length 1 from an object of the class `ObjectVar` with the first version of the monotonic class semantics discussed in Example 3:

```
ObjectVar a = new ObjectVar();
a.set(a);    // a cycle is built
b = a.get();
c = b.get(); // c != b
d = c.get(); // d != b && d!= c
```

Notice that to preserve monotonicity according to the above semantics, different references are returned in variable `a`, `b`, and `c`. Thus, we met a problem that, although the cycle was built, it can never be recognized programmatically, while traversing the object structure. For some applications this may be appropriate, for example, when a graph is the representation of a finite automata, which is used only in its interpreter. But if, for example, we wanted to print the automata representation, we would not be able to write a terminating code.

Example 5: a cyclic graph with a finite number of references to vertices. Although we don't know how to represent and traverse graphs having access to the unique references to edges in the world of monotonic objects as easily as we do in object-oriented languages, there are particular solutions to this problem. Consider two of them.

Example 5a: Imagine a factory method that simultaneously creates a given number of objects and returns an immutable vector of the references to them. Its signature may be like this:

```
Vector<ObjectVar> createObjectVars(int n);
```

Then let us modify the class `ObjectVar` in such a way that its objects "know brothers", that is, each object has access to this vector, and the `get` method returns only references to the "brothers" and to the object itself. Classes with such semantics allow us to build an efficient representation of an arbitrary finite graph. We argue that such class declarations are monotonic.

Example 5b: Let us return to the monotonic class declaration in Examples 3 and 4 and use it for further modification. Let us prohibit returning reference values from monotonic objects until the whole of the deep structure accessible from the given object is fully defined (and hence, is finite). Then let us minimize the graph representation by means of the well-known algorithm of minimization of a finite automaton. We argue that the unique references to the objects of the minimal representation can now be returned by the methods like `get`, preserving monotonicity. There are a finite number of references to the accessible objects and while traversing the graph structure, we can programmatically catch the cycles.

More parallelism by suspending the equality checks and exceptions. One more subtlety that can limit parallelism is that the method `equal` (in the `if` statement of Example 2) cannot return `true` until the compared objects become fully defined. Such blocking of the computation is highly undesirable. Fortunately, there is an escape. The definition of monotonicity does not distinguish various exceptions raised in different program points. All exceptions are equivalent as the result of computation. This gives us a possibility to suspend execution of such `equal` predicates along with the surrounding `if` statements and immediately return from the `set` method. If, in the end, when the compared objects become defined, `equal` returns `true`, the suspended statement completes with no

effect. If a difference is found and `equal` returns `false`, the exception is raised from the suspended statement and propagated as the resulting exception of the whole computation. This behavior is monotonic according to our definition.

6. Acknowledgement

We express our gratitude to our English teacher, Philippa Jephson, who helped us edit this paper, fix a lot of errors and turn it into a much more readable form. The remaining glitches are ours.

7. Conclusion

The notion of a monotonic class and a monotonic object was presented. It was defined so that the use of monotonic objects in a program in a functional-like language with maximal parallelism preserves the main properties of functional languages: determinism of computation results (confluence) and possibility of repeated computations with the same result and side effect (idempotency). We use the term *monotonic* with the idea that a monotonic object changes in a certain (semi)lattice that can be derived from its class declaration.

Unlike functional languages, building cyclic data structures is possible with the use of monotonic classes, although their semantics and code are nontrivial and tricky, and impose certain overheads compared to common object-oriented programming. Further development of the methods of their efficient implementation is required taking into consideration special cases and using metacomputation methods like program specialization. Currently, we are prototyping an implementation of a language with monotonic classes.

The demonstrated examples show that it is not at all obvious whether a class declaration is monotonic or not. Designing such a class is like finding a nontrivial solution to an “equation” that is the property of monotonicity. Formal means (theory and software tools) are highly desirable in helping us prove monotonicity. This is an interesting program verification topic for future work.

References

1. Abramov S.M., Adamovich A.I., Kovalenko M.R. T-System—An Environment Supporting Automatic Dynamic Parallelization of Programs: An Example of the Implementation of an Image Rendering Algorithm Based on the Ray Tracing Method // Programmirovaniye, 25:2. 1999. P. 100–107. (In Russian).
2. Adamovich A.I. Fibers as the Basis for the Implementation of the Notion of the T-Process for the JVM Platform // Program Systems: Theory and Applications, 6:4 (27). 2015. P. 177–195. URL: http://psta.psiras.ru/read/psta2015_4_177-195.pdf. (In Russian).

3. Adamovich A.I. The Ajl Programming Language: The Automatic Dynamic Parallelization for the JVM Platform // Program Systems: Theory and Applications, 7:4 (31). 2016. P. 83–117. URL: http://psta.psiras.ru/read/psta2015_4_177-195.pdf. (In Russian).
4. Adamovich A.I., Klimov And.V. On Experience of Using the Metaprogramming Development Environment Eclipse/TMF for Construction of Domain-Specific Languages // Nauchnyy servis v seti Internet, Trudy XVIII Vserossiyskoy nauchnoy konferentsii (September 19–24, 2016, Novorossiysk, Russia). Moscow: Keldysh Institute of Appl. Math. of RAS: 2016. P. 3–8. URL: http://keldysh.ru/abrau/2016/45.pdf. (In Russian).
5. Adamovich A.I., Klimov And.V. How to Create Deterministic by Construction Parallel Programs? Problem Statement and Survey of Related Works // Program Systems: Theory and Applications, 8:4(35). 2017. P. 221–244. URL: http://psta.psiras.ru/read/psta2017_4_221-244.pdf. (In Russian).
6. Bocchino R.L. (Jr.), Adve V.S., Adve S.V., Snir M. Parallel Programming Must Be Deterministic by Default // Fifth USENIX Conference on Hot Topics in Parallelism, HotPar'09. Berkeley, CA, USA: USENIX Association, 2009. P. 4–4.
7. Bocchino R.L. (Jr.), Adve V.S., Dig D., Adve S.V., Heumann S., Komuravelli R., Overbey J., Simmons P., Sung H., Vakilian M. A Type and Effect System for Deterministic Parallel Java // SIGPLAN Not., 44:10. 2009. P. 97–116.
8. Klimov And.V. Dynamic Specialization in Extended Functional Language with Monotone Objects // SIGPLAN Not., 26:9. 1991. P. 199–210.
9. Klimov And.V. Deterministic Parallel Computations with Monotonic Objects // Nauchnyy servis v seti Internet: mnogoyadernyy komp'yuternyy mir. 15 let RFFI, Trudy Vserossiyskoy nauchnoy konferentsii (24–29 sentyabrya 2007 g., g. Novorossiysk). Moscow: Izd-vo Moskovskogo universiteta, 2007. P. 212–217, URL: https://pat.keldysh.ru/~anklimov/papers/2007-Klimov--Deterministic_Parallel_Computation_with_Monotonic_Objects.pdf. (In Russian).
10. Marlow S. Parallel and Concurrent Programming in Haskell. CA, USA: O'Reilly, 2013.
11. Kawaguchi M., Rondon P., Bakst A., Jhala R. Deterministic Parallelism via Liquid Effects // ACM SIGPLAN Not., 47:6. 2012. P. 45–54.
12. Kuper L. Lattice-Based Data Structures for Deterministic Parallel and Distributed Programming, Ph.D. Thesis. 2015. URL: https://users.soe.ucsc.edu/~lkuper/papers/lindsey-kuper-dissertation.pdf.
13. Kuper L., Todd A., Tobin-Hochstadt S., Newton R.R. Taming the Parallel Effect Zoo: Extensible Deterministic Parallelism with LVish // ACM SIGPLAN Not., 49:6. 2014. P. 2–14.
14. Kuper L., Turon A., Krishnaswami N.R., Newton R.R. Freeze after Writing: Quasi-Deterministic Parallel Programming with LVars // ACM SIGPLAN Not., 49:1. 2014. P. 257–270.
15. Kuper L., Newton R.R. LVars: Lattice-Based Data Structures for Deterministic Parallelism // 2nd ACM SIGPLAN Workshop on Functional High-performance Computing, FHPC'13. New York, NY, USA: ACM, 2013. P. 71–84.