Concurrency: from a Calculus to Objects

Extended Abstract

Martin Henz, Martin Müller, Christian Schulte

German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany

email: {henz,mmueller,schulte}@dfki.uni-sb.de

June 1, 1995

Summary

Our goal is the design of a concurrent object-oriented programming language. It has been shown to be useful in programming language design to start with a *formal* system describing the computation. On the other hand, *intuitive* programming abstractions are crucial for a useful programming language. Concurrent objects seem to be the most useful intuition for concurrent computation. In this paper, we show how we use a formal system for concurrent computation, called the Oz Calculus, to express a programming style that follows intuitions of concurrent objects. We explain the advantages of this calculus for concurrent objects over other approaches and how these advantages can be used in the concurrent object-oriented programming language Oz.

1 Towards a Calculus for Concurrent Objects

It is widely accepted in the programming language community that a programming language should be built on top of a simple yet expressive formal calculus. Besides of being the base for a rigorous language definition, a formal calculus eases the development of implementation technologies and provides a good starting point for reasoning about programs. Programming abstractions can be composed from entities of the formal model. This approach has been pioneered by functional programming languages (see for instance [10]), which are usually built on top of the λ-calculus.

Surprisingly, large parts of the research in concurrent object-oriented programming (coop) did not attempt to start from such a formal base for computation. Instead, intuitions on concurrent objects have been used to guide the design and implementation of their programming languages [1, 18].

To our knowledge, the first attempt to build a coop language on top of a formal calculus has been made in the concurrent constraint programming language Oz [3], which is based on a formal model [15, 16] extending the concurrent constraint framework [13]. Other attempts are described in [11], where Milner's π-calculus [9] has been used as a base for designing the concurrent object-oriented language PICT, and in [17], where a typed calculus for concurrent objects is used as a base for a simple coop language.

In contrast to most others, the Oz approach is not based on message passing for communication between concurrent activities, but instead adheres to a shared memory model.

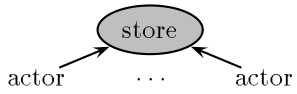
Since the first formal system for concurrent computation in [5], shared memory was ruled out as a viable base for concurrency. However, shared memory in form of a a monotonically growing constraint store as proposed by concurrent constraint (`cc`) languages [7, 13] seems to provide a superior means for communication in a concurrent setting.

Although the usual `cc` languages [2] are able to express concurrent computation, they are lacking two ingredients to express powerful `coop` programming: higher-order programming and mutable state. The Oz Calculus [16] extends `cc` by these features and serves us as a base for `coop` programming.

2 Properties of the Oz Calculus

In this section we will show some properties of computation in the Oz Calculus. These properties are important for how concurrent objects are handled in the resulting programming language Oz.

Actors. Computation proceeds by reducing actors connected to a store. An actor reduces as soon as the store contains sufficient information. Actors perform computational tasks like synchronization, procedure definition and application, variable creation, and adding of information to the store. By performing its task the actor ceases to exist and may create new actors.



The Store. The store holds information on data structures that the actors have access to by means of variables. The information grows monotonically over time. Incrementally and concurrently, a data structure can be constructed by several actors. Technically, the store consists of constraints describing the values of the variables, which are often referred to as logic variables.

Synchronization and Communication. Synchronization actors can specify the amount of information they require to compute. For instance, an actor can write a cons cell with not yet specified content to the store. A second actor can synchronize on the data structure to be a cons cell, and in turn can create actors to add information on the element of the cons cell. By doing so, actors communicate through their common store. Synchronization on information is implemented by checking entailment of constraints.

Threads. For writing fair and efficient programs, actors are grouped into threads. A thread is a stack of actors, reduction of actors follows the structure of the thread. If a thread encounters a actor which has not enough information to proceed, the actor is moved to a newly created thread.

Full Compositionality. Actors can perform procedure definition. Together with static scoping of variables, this allows to provide all higher-order programming techniques used in functional programming languages.

3 Objects

The Oz calculus is able to express the Actor model of computation [4]. Furthermore, unifying synchronization and concurrency with the notion of a constraint store and full compositionality opens up a realm of further programming techniques:

Incomplete Messages. In the computation model actors communicate through their common store. Message sending to an object amounts to an actor which writes the message to the store and invokes an actor which reads the message. As argued above, both writer and reader can add new information to the message. Message arguments, return values and futures are just special cases of this general scheme, known as "incomplete messages" [14].

Active objects are as passive as possible. Message sending amounts to invoking an actor which reads the message from the store. This invocation continues to compute on the sending thread as long as it has enough information. Only when information is lacking, as is the case for busy active objects, a new thread is created. Objects remain passive and highly efficient as long as they are used sequentially, automatically become active when needed, and return to be passive whenever possible. Thus, sophisticated compile-time analysis as in [12] to detect cases where the overhead of active objects are removed, becomes obsolete in Oz.

Fine-grained Concurrency. Fine-grained concurrency, together with ubiquitous synchronization allows to restrict standard object protocols such that no constraints on the acceptance of messages are imposed. I.e. every object has to accept every message. This simplifies object model and implementation and eliminates the source of the inheritance anomaly [8] inherent in many `coop` languages.

Many-to-one Communication. Mutable state in the Oz Calculus allows to directly express many-to-one communication, as opposed to other `cc` languages in which streams [14] are used for communication. Streams make many-to-one communication hard to achieve [6], are memory consuming and do not leave active objects as passive as possible.

Inheritance and Parameterization. Higher-order programming is useful to express (multiple) inheritance and a flexible object protocol. Methods of objects are represented by procedures. Since procedures are first-class values, classes can be represented by data structures holding the methods of their instances. Inheritance can be accommodated by Oz procedures computing new classes from existing ones. Object protocols can be written in Oz and inherited so that experimentation is possible without compiler modification. Classes, objects, messages, message and attribute names can be dynamically created and passed around.

References

[1] G. Agha, P. Wegner, and A. Yonezawa, editors. *Research Directions in Object-Based Concurrency.* MIT Press, Cambridge, MA, 1993.

[2] A. Davison. A survey of logic programming-based object oriented languages. In *Research Directions in Concurrent Object-Oriented Programming.* The MIT Press, Cambridge, MA, 1993.

[3] M. Henz, G. Smolka, and J. Würtz. Object-oriented concurrent constraint programming in Oz. In V. Saraswat and P. V. Hentenryck, editors, *Principles and Practice of Constraint Programming*, chapter 2, pages 27–48. The MIT Press, Cambridge, MA, 1995. To appear.

[4] C. Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Artificial Intelligence*, 8:323–364, 1977.

[5] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[6] K. Kahn. Objects: A fresh look. In *Proceedings of the European Conference on Object Oriented Programming*, pages 207–223, 1989.

[7] M. J. Maher. Logic semantics for a class of committed-choice programs. In J.-L. Lassez, editor, *Logic Programming, Proceedings of the Fourth International Conference*, pages 858–876. The MIT Press, 1987.

[8] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Based Concurrency*. MIT Press, 1993.

[9] R. Milner. The polyadic π-calculus: A tutorial. ECS-LFCS Report Series 91-180, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh EH9 3JZ, Oct. 1991.

[10] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. International Series in Computer Science. Prentice-Hall, 1987.

[11] B. C. Pierce and D. N. Turner. Concurrent Objects in a Process Calculus. In *Theory and Practice of Parallel Programming (TPPP)*, Lecture Notes in Computer Science, Sendai, Japan, Nov. 1994. Springer-Verlag, Berlin, Germany.

[12] J. Plevyak, X. Zhang, and A. Chien. Obtaining sequential efficiency for concurrent object-oriented languages. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 311–321, San Francisco, 1995. ACM Press.

[13] V. A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages*, pages 232–245, San Francisco, CA, Jan. 1990.

[14] E. Shapiro and A. Takeuchi. Object oriented programming in Concurrent Prolog. *New Generation Computing*, 1:24–48, 1983.

[15] G. Smolka. A foundation for higher-order concurrent constraint programming. In J.-P. Jouannaud, editor, *1st International Conference on Constraints in Computational Logics*, Lecture Notes in Computer Science, vol. 845, pages 50–72, München, Germany, 7–9 Sept. 1994. Springer-Verlag.

[16] G. Smolka. The Definition of Kernel Oz. In A. Podelski, editor, *Constraints: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 251–292. Springer-Verlag, Berlin, Germany, 1995.

[17] V. T. Vasconcelos. Typed Concurrent Objects. In *Proceedings of the European Conference on Object Oriented Programming*, pages 100–117, 1994.

[18] A. Yonezawa and M. Tokoro, editors. *Object-Oriented Concurrent Programming*. The MIT Press, Cambridge, MA, 1987.