Objects in Higher-order Concurrent Constraint Programing with State

Martin Henz and **Gert Smolka**
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, D–66123 Saarbrücken, Germany
{henz,smolka}@dfki.uni-sb.de
Fax: +49 681 302-5341, Phone: +49 681 302-5311

Abstract

Concurrent constraint programming (ccp) languages allow to express concurrency on a clean semantic base. Since objects are an attractive abstraction for describing concurrent activity, it was the goal of several ccp languages to provide object-oriented programming (oop). The traditional way to express the notion of objects in ccp languages is by describing an object as the consumer of a communication medium, e.g. a message stream, bag, channel, or port.

We propose a new model for objects in higher-order ccp languages like Oz: Objects are (dynamically created) procedures that take messages as arguments. While in previous proposals for objects in Oz a communication medium was still involved, we show in this paper that the right notion of concurrent state suffices to express oop in ccp. In contrast to the traditional approach, where communication, synchronization and buffering messages are all provided by the communication medium, the new model expresses them by orthogonal language constructs, allowing for a cleaner and more flexible conceptual base for objects, and a more efficient implementation in the case of light-weight objects.

Keywords Concurrent constraint programming, higher-order programming, object-oriented programming, state, Oz.

1 Introduction

In this section, we give a brief overview of previous approaches to objects in concurrent logic and ccp languages. A more comprehensive overview provide Haridi, Janson and Montelius in [5].

Since the first time when objects were expressed in a concurrent logic programming language by Shapiro and Takeuchi [10], research concentrates on finding the right communication medium. In this seminal paper, lists, also called *streams* in this context, serve as medium. Messages are sent by incrementally instantiating a stream. The receiver reads the next message from the stream, computes a new state with which it calls itself recursively using the tail of the stream. For example, a counter object is expressed by the following Concurrent Prolog program:

```
counter( [clear|S],State) ←
    counter(S,0).
counter( [inc|S],State) ←
    plus(State,1,NewState),counter(S?,NewState?).
counter( [get(State)|S],State) ←
    counter(S,State).
counter([],State).
```

A key question in all attempts towards objects in such languages is how to express many-to-1 communication. In the case of streams, there are two options:

- Each sender has its private stream to the receiver. All streams to an object are merged together into one stream which is consumed by the receiver.
- Languages with atomic test-and-unify allow multiple writers to a single stream.

Both approaches suffer efficiency problems as pointed out in [5]. Attempts to overcome these insufficiencies include *mutual references* [9], channels [15], and bags [6].

For the ccp language AKL [4], which combines Prolog-style search-oriented nondeterministic computation with process-oriented committed-choice and ccp, Haridi, Janson and Montelius [5] propose the concept of ports. A port is a connection between a bag of messages which serves as the object identity for the sender and a stream that provides the receiver with the access to the received messages.

Common to all approaches is the emphasis on the communication medium, which serves three orthogonal purposes:

- communication and identification: The medium links the sender and receiver and provides the sender with a unique identity of the receiver.
- synchronization: Many-to-1 communication is enabled by allowing the receiver to nondeterministically choose one of the incoming messages while leaving the others waiting in the medium.
- buffering: Messages are stored in the medium until the receiver, residing in a different branch of concurrent computation in form of a suspending clause, is ready to pick them up. This incurs an inevitable memory overhead, and may cause a run-time penalty due to context switching in sequential implementations.

In the new model for oop in Oz, the ubiquitous concept of defining and calling procedures serves the first purpose by encoding object creation by procedure definition and message sending by procedure call.

For the second purpose, we introduce the concept of a *concurrent memory cell.* This is a significant simplification over previous proposals for objects in Oz in [12] and [3].

The third purpose becomes obsolete since messages that cannot be served are represented by suspending applications of the object. Sequential object-oriented programming can be supported especially efficiently, since in this case no such suspensions occur.

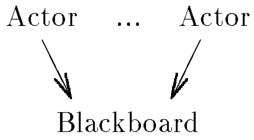
After introducing a sublanguage of Oz in Section 2, and a couple of examples in Section 3, we will show in Section 4 that, with the additional notion of a concurrent memory cell, object-oriented programming including multiple inheritance, *self*, late method binding, private methods and attributes, encapsulation and object identity can be expressed by the means of the underlying language Oz.

2 A Sublanguage of Oz

This section gives an informal presentation of the basic computation model underlying a sublanguage of Oz that suffices for the purpose of this paper[1] (see [11] for a formal presentation). The full language and programming system of Oz is described in [2].

2.1 The Computation Space

Oz generalizes the model of ccp [7] by providing for higher-order programming. Central to the computation model of Oz is the notion of the *computation space.* It consists of a number of actors[2] connected to a blackboard.



The actors read the blackboard and reduce once the blackboard contains sufficient information. When an actor reduces, it may put new information on the blackboard and create new actors. As long as an actor does not reduce, it does not have an outside effect. The actors of the computation space are short-lived: once they reduce they disappear.

The blackboard stores a constraint (constraints are closed under conjunction, hence one constraint suffices) and a number of named abstractions (to

[1] We omit deep guard computation, disjunction, encapsulated search, and finite domain constraints.

[2] Oz's actors are different from Hewitt's actors. We reserve the term agent for longer-lived computational activities enjoying persistent and first-class identity.

be explained later). Constraints are formulae of first-order predicate logic with equality that are interpreted in a fixed first-order structure called the universe. The universe provides rational trees as in Prolog II and records (see Section 2.3). The constraint on the blackboard is always satisfiable in the universe and becomes monotonically stronger over time. We say that a blackboard entails a constraint ψ if the implication $\phi \rightarrow \psi$ is valid in the universe, where ϕ is the constraint stored on the blackboard. We say that a blackboard is consistent with a constraint ψ if the conjunction $\phi \wedge \psi$ is satisfiable in the universe, where ϕ is the constraint stored on the blackboard. Since the constraint on the blackboard can only be observed through entailment and consistency testing, it suffices to represent it modulo logical equivalence.

2.2 Elaboration of Expressions

There are several kinds of actors. This section will introduce elaborators and conditionals.

An elaborator is an actor executing an expression. The expressions we will consider in this section are defined as follows:

$$
\begin{array}{lcl}
E, F & ::= & \phi \quad | \quad E\ F \quad | \quad \mathbf{local}\ x\ \mathbf{in}\ E\ \mathbf{end} \\
 & | & \mathbf{proc}\ \{x\ y_1 \ldots y_n\}\ E\ \mathbf{end} \quad | \quad \{x\ y_1 \ldots y_n\} \\
 & | & \mathbf{if}\ C_1 \,\square\, \ldots \,\square\, C_n\ \mathbf{else}\ E\ \mathbf{fi} \\
C & ::= & x_1 \ldots x_n\ \mathbf{in}\ \phi\ \mathbf{then}\ E
\end{array}
$$

Elaboration of a constraint ϕ checks whether ϕ is consistent with the blackboard. If this is the case, ϕ is conjoined to the constraint on the blackboard; otherwise, an error is reported. Elaboration of a constraint corresponds to the eventual tell operation of ccp.

Elaboration of a concurrent composition $E\ F$ creates two separate elaborators for E and F.

Elaboration of a variable declaration **local** x **in** E **end** creates a new variable (local to the computation space) and an elaborator for the expression E. Within the expression E the new variable is referred to by x. Every computation space maintains a finite set of local variables.

Elaboration of a procedure definition **proc** $\{x\ y_1 \ldots y_n\}$ E **end** chooses a fresh name a, writes the named abstraction $a\colon y_1 \ldots y_n/E$ on the blackboard, and creates an elaborator for the constraint $x = a$. Names are constants denoting pairwise distinct elements of the universe; there are infinitely many of them. Since abstractions are associated with fresh names when they are written on the blackboard, a name cannot refer to more than one abstraction.

Elaboration of a procedure application $\{x\ y_1\ \ldots\ y_n\}$ waits until the blackboard entails $x = a$ and contains a named abstraction $a\colon x_1 \ldots x_n/E$, for some name a. When this is the case, an elaborator for the expression $E[y_1/x_1 \ldots y_n/x_n]$ is created ($E[y_1/x_1 \ldots y_n/x_n]$ is obtained from E

by replacing the formal arguments $x_1, \ldots, x_n$ with the actual arguments $y_1, \ldots, y_n$).

This simple treatment of procedures provides for all higher-order programming techniques. By making variables denote names rather than higher-order values, we obtain a smooth combination of first-order constraints with higher-order programming.

Elaboration of a conditional expression of the form **if** $\overline{x}_1$ **in** ϕ_1 **then** E_1 □ ... □ $\overline{x}_n$ **in** ϕ_n **then** E_n **else** F **fi** creates a conditional actor, which waits until the blackboard either entails one of $\exists \overline{x}_i \phi_i$, in which case the other clauses are discarded and conditional actor reduces to an elaborator for **local** $\overline{x}_i$ ϕ_i E_i **end**, or all ϕ_i are disentailed, in which case the conditional actor reduces to an elaborator for F.

2.3 Records as Logical Data Structure

Records are the congenial data structure for object-oriented programming. Oz provides records with a set of constraints operating on records [13] [1]. A constraint of the form

$$X = a(a_1 : t_1 \ldots a_n : t_n)$$

means that X a record labeled by the constant a (called its label), whose n fields containing terms t_i are accessible through the features a_i, $i \leq n$. In the sequel, we use three additional constraints[3] to compute with records:

- R.F=X is true, if the record R contains the term X in the field with descriptor F. For example, the constraint Y=point(x:1 y:2).y is equivalent to Y=2.

- {Adjoin R1 R2 R3} is true if R3 is equal to the record R1, except that for equal features the field of R2 overrides the field of R1 and for disjoint features the field is added. For example, the constraint {Adjoin point(x:1 y:2) point(x:3 color:red) Z} is equivalent to Z=point(x:3 y:2 color:red).

- {AdjoinAt R1 F X R2} is true if R2 is equal to R1, except that R2 has the field X in feature F. For example, the constraint {AdjoinAt point(x:1 y:2) y:5 Z} is equivalent to Z=point(x:1 y:5).

3 Examples

The following example program shows the interplay of procedure definitions, applications and conditionals.

[3] Strictly speaking, they are not constraints but actors that wait for the first two arguments to become determined.

```
declare Length in
proc {Length Xs N}
   if Xs=nil then N=0
   [] X Xr M in Xs=X|Xr then N=s(M) {Length Xr M}
   else false fi
end
```

The **declare** expression is a variant of the **local** expression whose scope extends to expressions the programmer enters later. Elaboration of the procedure definition chooses a new name a, writes the named abstraction a: Xs N / **if...fi** on the blackboard and binds Length to a.

The expression

```
declare Xs Xr N in Xs=_|Xr {Length Xs N}
```

is elaborated as follows. The constraint Xs=_|Xr is put on the blackboard[4] and the application is replaced by the body of the procedure. This conditional will reduce to its second clause since the constraint ∃ X Xr M Xs=X|Xr is entailed by the current blackboard. Thus, the expression **local** X Xr M **in** Xs=X|Xr N=s(M) {Length Xr M} **end** will be elaborated. After one further unfolding of the procedure Length the resulting conditional actor will suspend for lack of information on Xr. At this point, the variable N is constrained to s(M) with no information on M. Only after Xs becomes a list of known length for example by

```
Xr=1|_|nil
```

the length of the list N can be computed to s(s(s(0))).

The next example shows how procedures are dynamically created. Consider the expression

```
declare MakeAdder in
proc {MakeAdder N Adder}
   proc {Adder Y Z}
      Z=N + Y
   end
end
```

Elaboration of

```
declare A in
local A4 in {MakeAdder 4 Adder4} {Adder4 1 A} end
```

results in unfolding the procedure MakeAdder. Elaboration of the procedure definition of Adder chooses a new name a, writes the named abstraction a:Y Z / Z=N + Y on the blackboard and binds Adder4 to a. The application {Adder4 1 A} can then be unfolded as usual, binding A to 5.

[4]Like in Prolog, the symbol _ denotes a new and anonymous variable.

4 Objects in Oz

4.1 The Traditional Way

Before we introduce the new approach in Section 4.4, we present in Program 4.1, how stream-based communication is achieved in Oz.

Note that the Oz compiler is able to expand functional notation; for example the expression {MethodTable.{Label Message} State Message NewState} expands to

```
local X Y in
   {Label Message X}
   MethodTable.X=Y
   {Y State Message NewState}
end
```

The application {Label Message X} binds X to the label of the tree or record Message. For example, {Label get(Y) X} binds X to get and {Label inc X} binds X to inc.

Program 4.1 Stream-based Object in Oz

```
declare MethodTable Feed in
MethodTable=
methods(clear: proc {$ InState clear OutState}
                  OutState={AdjoinAt InState val 0}
               end
        inc: proc {$ InState inc OutState}
                OutState={AdjoinAt InState val InState.val + 1}
             end
        get: proc {$ InState get(X) OutState}
                OutState=InState
                X=InState.val
             end
       )
proc {Feed Stream State}
   if Message NewStream in
     Stream=Message|NewStream
   then
      local NewState in
         {MethodTable.{Label Message} State Message NewState}
         {Feed NewStream NewState}
      end
   fi
end
```

The variable MethodTable is bound to a record in which the methods (procedures) are stored that the object applies when it receives a message. The procedure Feed represents the object protocol. When it is called like in

```
declare S in {Feed S state(val:0)}
```

it suspends until the stream S becomes constrained to Message|NewStream. Suppose this is done by

```
declare S1 in S=inc|S1
```

Then the method table is accessed using the label of the message, in this case the atom inc, resulting in the inc procedure. This procedure is applied to state(val:0) and inc and NewState, resulting in binding NewState to state(val:1). The protocol Feed is called recursively with S1 and NewState.

4.2 Discussion

Common to all proposed solution mentioned in the introduction is the medium linking the sender with the receiver, exemplified by the variable Stream. The sender leaves its message on the stream and proceeds. This incurs an inevitable memory overhead, in this case a list construction. Furthermore, for the receiver to deal with the message, a context switch is necessary in sequential implementations. Clearly, for light-weight objects, such as window components, one would prefer to execute the appropriate method on the sender's side without suspension or context switch. Our goal is to efficiently support light-weight objects and switch to a communication medium like ports only for heavy weight (distributed) objects where it is often cheaper to transfer the message to the receiver than the method including the object's state to the sender.

Already in previous proposals for objects in Oz [12] and [3], methods were executed on the sender's side. However, there was still a communication medium involved. With an appropriate notion of concurrent state described in the next section, we can significantly simplify the model.

4.3 The Concurrent Memory Cell

In this section, we describe the concept of concurrent memory cells that is used for objects in the next section. There are two primitive operations to support memory cells. The first operation

```
{NewCell C X}
```

creates a new cell C with content X. Like in procedure definition, a new name a is chosen and the constraint C=a becomes elaborated. The named memory cell a:X is written on the blackboard.

The second operation allows to simultaneously to read and write the cell.

```
{ExchangeCell C OldValue NewValue}
```

waits until the blackboard entails C=a and contains a named memory cell a:X. When this is the case, the constraint OldValue=X becomes elaborated and the memory cell is replaced by a:NewValue. This is an atomic operation.

For example, in

```
declare C in
local Value in {NewCell C 1} {ExchangeCell C Value 2} end
```

the variable Value will be bound to 1 and when the cell is accessed a second time like in

```
local SecondValue in {ExchangeCell C SecondValue 2} end
```

the variable SecondValue will be bound to 2.

4.4 Objects with Memory Cells

Program 4.2 shows an implementation of the counter object based on memory cells.

Program 4.2 Cell-based Object in Oz

```
declare Counter in
local C in
   {NewCell C state(val:0)}
   proc {Counter Message}
      local State NewState in
         {ExchangeCell C State NewState}
         {MethodTable.{Label Message} State NewState}
      end
   end
end
```

The object identity is provided by the procedure Counter, represented by a named abstraction on the blackboard. The state of the object is stored in the cell C which initially contains the record state(val:0). Communication is provided by application. For example the message sending

```
{Counter inc}
```

results in exchanging the state by a new variable NewState. Now, the appropriate method is applied resulting in binding NewState to state(val:1).

The computation resulting from message sending is performed on the sender's side. If the state is available (e.g. because it resides on the same node), the method can be directrly applied in the style of sequential object-oriented programming.

4.5 Light-weight vs. Distributed Objects

Light weight objects like window components can be implemented efficiently using memory cells.

In parallel and distributed programming, objects residing on different nodes of a network may communicate. Clearly, it could be disadvantagous to transfer both the receiver's state and the appropriate method to the sender instead of transferring the message to the receiver as is the case in the traditional model.

For these cases, we switch to the communication medium of ports. Program 4.3 show how ports can be implemented using memory cells. The

Program 4.3 Implementing Ports with Cells

```
proc {MakePort O ?P}
   local
      Stream Cell
   in
      Cell = {NewCell Stream}
      proc {P X}
         local Tail in {ExchangeCell Cell X|Tail Tail} end
      end
      {ForAll Stream O}
   end
end
```

procedure MakePort creates a port for a procedure O in form of the procedure P. If P is called, its argument X is entered in the port by instantiating the current content of the cell Cell to a list with X as first element. The new content of Cell is the tail of the list. The procedure O is called on every element of the thereby growing stream.

For example, **declare** LocalCounter **in** {MakePort Counter LocalCounter} creates a local "worker" for the object Counter. The computation triggered by a message sending like {LocalCounter inc} is performed where the port was made, and not where the message was sent.

4.6 Garbage Collection

In [5], a main issue is the garbage collection of the communication medium. Since in Oz, the communication medium is provided by the general mechanism of defining and calling procedures, there is no memory overhead for communication. Like any other procedure, an object is garbage-collected when there is no reference to it anymore. Objects can also be explicitly closed which allows to garbage-collect the object's state even if there are still references to the object. Further messages to a closed object are ignored.

4.7 Self and Multiple Inheritance

The concept of *self* is provided by adding a further argument to the methods as in the following fragment of a method table

```
inc2:proc {$ InState inc2 Self OutState}
        {Self inc}
        {Self inc}
     end
```

The object protocol ensures that methods are always called with the receiving object as third argument.

Object definition by multiple inheritance is performed by adjoining the method tables of parent objects, overriding methods using a precedence, defined by the algorithm of class precedence lists of CLOS [14].

In Oz, some syntactic sugar is provided to allow to express oop more elegantly. The Counter object in Program 4.2 can be written as

```
create Counter from UrObject
   attr val:0
   meth clear val ← 0 end
   meth inc   val ← @val + 1 end
   meth get(X) X=@val end
end
```

An object Counter2 that contains additionally the method inc2 above can be created by inheritance as in

```
create Counter2 from Counter
   meth inc2 {self inc} {self inc} end
end
```

Details of the expansion of this syntax can be found in [2].

5 Conclusion

We showed that the overhead of a communication medium for oop in ccp languages for light-weight objects can be overcome in the higher-order ccp language Oz by introducing the concept of a concurrent memory cell.

For distributed applications, we can switch to ports which can be efficiently implemented using concurrent memory cells.

In the resulting concept of objects in Oz, objects can be embedded in arbitrary data structures, including messages and the state of other objects. Since objects and methods are procedures, they can be created within other objects and procedures and can, vice versa, create objects and procedures themselves. Since messages are terms, they can contain logical variables and provide objects with the full range of constraint programming techniques such as incomplete messages.

Acknowledgements and Remark

The research reported in this paper has been supported by the Bundesminister für Forschung und Technologie (FTZ-ITW-9105), the Esprit Project ACCLAIM (PE 7195), and the Esprit Working Group CCL (EP 6028).

The Oz System and its documentation are available through anonymous ftp from `duck.dfki.uni-sb.de` or through WWW from `http://www.dfki.uni-sb.de/`.

References

[1] H. Aït-Kaci, A. Podelski, and G. Smolka. A feature-based constraint system for logic programming with entailment. *Journal of Theoretical Computer Science*, 122(1–2):263–283, Jan. 1994.

[2] M. Henz, M. Mehl, M. Müller, T. Müller, J. Niehren, R. Scheidhauer, C. Schulte, G. Smolka, R. Treinen, and J. Würtz. The Oz Handbook. Research Report RR-94-09, DFKI, 1994. Available through anonymous ftp from `duck.dfki.uni-sb.de`.

[3] M. Henz, G. Smolka, and J. Würtz. Oz—a programming language for multi-agent systems. In *13th International Joint Conference on Artificial Intelligence*, volume 1, pages 404–409, Chambéry, France, 1993. Morgan Kaufmann Publishers.

[4] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *International Logic Programming Symposium*, pages 167–186, 1991.

[5] S. Janson, J. Montelius, and S. Haridi. Ports for objects. In *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press, Cambridge, Mass., 1993.

[6] K. M. Kahn and V. A. Saraswat. Actors as a Special Case of Concurrent Constraint Programming. In *Proceedings of the European Conference on Object Oriented Programming*, pages 57–66. ACM, October 1990.

[7] V. 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, January 1990.

[8] E. Shapiro, editor. *Concurrent Prolog. Collected Papers. Volume 1 and 2.* MIT Press, 1987.

[9] E. Shapiro and S. Safra. Multiway merge with constant delay inConcurrent Prolog. In E. Shapiro, editor, *[8]*, chapter 15, pages 414–420. MIT Press, 1987.

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

[11] G. Smolka. A calculus for higher-order concurrent constraint programming with deep guards. Research Report RR-94-03, DFKI, Feb. 1994. Available through anonymous ftp from `duck.dfki.uni-sb.de`.

[12] G. Smolka, M. Henz, and J. Würtz. Object-oriented concurrent constraint programming in Oz. Research Report RR-93-16, DFKI, April 1993. Will appear in: P. van Hentenryck and V. Saraswat (eds.), Principles and Practice of Constraint Programming, The MIT Press, Cambridge, Mass.

[13] G. Smolka and R. Treinen. Records for logic programming. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 240–254. The MIT Press, 1992. Full version has appeared in Journal of Logic Programming, April 1994.

[14] G. L. Steele. *Common LISP. The Language. 2nd Ed.* Digital Press, 1990.

[15] E. Tribble, M. S. Miller, K. Kahn, B. D. G., and C. Abott. Channels: A Generalization of Streams. In E. Shapiro, editor, *[8]*, chapter 17, pages 446–463. MIT Press, 1987.