1 Object-Oriented Concurrent Constraint Programming in Oz*

Martin Henz, Gert Smolka, and Jörg Würtz

1.1 Abstract

Oz is a higher-order concurrent constraint programming system under development at DFKI. It combines ideas from logic and concurrent programming in a simple yet expressive language. From logic programming Oz inherits logic variables and logic data structures, which provide for a programming style where partial information about the values of variables is imposed concurrently and incrementally. A novel feature of Oz is the support of higher-order programming without sacrificing that denotation and equality of variables are captured by first-order logic. Another new feature of Oz are cells, a concurrent construct providing a minimal form of state fully compatible with logic data structures. These two features allow to express objects as procedures with state, avoiding the problems of stream communication, the conventional communication mechanism employed in concurrent logic programming.

Based on cells and higher-order programming, Oz readily supports concurrent object-oriented programming including object identity, late method binding, multiple inheritance, "self", "super", batches, synchronous and asynchronous communication.

1.2 Introduction

Oz [6, 21, 20, 17, 16, 7] is an attempt to create a high-level concurrent programming language bringing together the merits of logic and object-oriented programming.

Our starting point was concurrent constraint programming [14], which brings together ideas from constraint and concurrent logic programming. Constraint logic programming [8, 1], on the one hand, originated with Prolog II [4] and was prompted by the need to integrate numbers and

*appeared in:
P. van Hentenryck and V. Saraswat (eds.), Principles and Practice of Constraint Programming, Chapter 2, pp. 29–48, The MIT Press, Cambridge, Mass.
Previous versions appeared as DFKI Research Report RR-93-16, April 1993, in the 1993 Conference on Programación Declarativa, in the 1993 Fachtagung für Künstliche Intelligenz, and as "Oz - A Programming Language for Multi-Agent Systems" in the 1993 International Joint Conference on Artificial Intelligence.

data structures in an operationally efficient, yet logically sound manner. Concurrent logic programming [18], on the other hand, originated with the Relational Language [3] and was promoted by the Japanese Fifth Generation Project, where logic programming was conceived as the basic system programming language and thus had to account for concurrency, synchronization and indeterminism. For this purpose, the conventional SLD-resolution scheme had to be replaced with a new computation model based on the notion of committed choice. At first, the new model was an ad hoc construction, but finally Maher [11] realized that commitment of agents can be captured logically as constraint entailment. A major landmark in the new field of concurrent constraint programming is AKL [9], the first implemented concurrent constraint language combining encapsulated search with committed choice.

The concurrent constraint model [14] can accommodate object-oriented programming along the lines of Shapiro and Takeuchi's stream-based model for Concurrent Prolog [19, 10]. Unfortunately, this model is intolerably low-level, which becomes fully apparent when one considers inheritance [5]. Vulcan, Polka, and A'UM are attempts to create high-level object-oriented languages on top of concurrent logic languages (see [10] for references). Due to the wide gap these languages have to bridge, they however lose the simplicity and flexibility of the underlying base languages.

Oz avoids these difficulties by extending the concurrent constraint model with the features needed for a high-level object model: a higher-order programming facility and a primitive to express concurrent state. With these extensions the need for a separate object-oriented language disappears, since the base language itself can express objects and inheritance in a concise and elegant way.

The way Oz provides for higher-order programming is unique in that denotation and equality of variables are nevertheless captured by first-order logic only. In fact, denotation of variables and the facility for higher-order programming are completely orthogonal concepts in Oz. This is in contrast to existing approaches to higher-order logic programming [13, 2].

Cells are a concurrent construct providing a minimal form of state fully compatible with constraints. They simply model a mutable binding of a name to a value, which can be changed by an atomic operation combining reading and writing.

Oz is based on a formal computation model accommodating concurrent computation as rewriting of a class of expressions modulo a structural congruence. This setup is known from a recent version of Milner's π-calculus [12]. It proves particularly useful for concurrent constraint computation since the structural congruence can elegantly model propagation and simplification of constraints.

Oz is fully implemented including garbage collection, incremental compilation and a window system based on Tcl/Tk. In terms of efficiency, it is competitive with emulated Sicstus Prolog. The Oz System and its documentation can be obtained via ftp from `ps-ftp.dfki.uni-sb.de` or through WWW from `http://ps-www.dfki.uni-sb.de/`.

A novel feature of Oz not treated in this paper is a higher-order combinator providing for encapsulated search [16, 17]. The search combinator allows to program different search strategies, including depth first and best solution (branch and bound) search.

The paper is organized as follows. Section 1.3 shows how the constraint system of Oz accommodates records. Section 1.4 gives an informal account of the computation model underlying Oz. Section 1.5 introduces the concrete language. Section 1.6 shows how objects can be modeled in Oz.

1.3 The Oz Universe

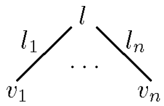
This section describes a fragment of the Oz Universe that suffices for the purpose of this paper.

The **Oz Universe** is a mathematical model of the data structures Oz computes with. It is defined as a structure of first-order predicate logic with equality. All variables in Oz range over the elements of the Oz Universe, called **values**. First-order formulas over its signature are called **constraints**. The value we consider are records and integers.

We describe the semantics of records informally; the mathematical details of the underlying construction are given in [22].

Records are composed using literals, denoted by l. A literal is either an atom or a name. An atom is a string (e.g., `val`, `get`). Names do not have a concrete syntax in Oz. It suffices to know that there are infinitely many names.

A **record** is either a literal or a proper record. A **proper record** is an unordered tree



where l is a literal, $l_1, \ldots, l_n$ are pairwise distinct literals, $v_1, \ldots, v_n$ are values, and $n > 0$.

Records are written as $l(l_1 \colon v_1 \ldots l_n \colon v_n)$, $n \geq 0$, where $l()$ stands for l. Two proper records are equal if and only if they have the same linear notation up to permutation of named fields $l_i \colon v_i$.

Given a record t of the form $l(l_1 \colon v_1 \ldots l_n \colon v_n)$, we call the literal l the **label**, the values $v_1, \ldots, v_n$ the **fields**, the integer n the **width**, and the literals $l_1, \ldots, l_n$ the **features** of t. Moreover, we call v_i the **field or subtree of t at l_i**.

An important operation on records is adjunction. The **adjunction** of two records s and t is the record $s * t$ defined as follows: the label of $s * t$ is the label of t; the features of $s * t$ are the features of s together with the features of t; and v is the subtree of $s * t$ at l if and only if either v is the subtree of t at l, or if l is not a feature of t and v is the subtree of s at l. Thus record adjunction amounts to record concatenation, where for shared features the right argument takes priority. For instance, the adjunction $l(a \colon 1\ b \colon 2\ c \colon 3) \ * \ k(b \colon 77\ d \colon 4)$ results in $k(a \colon 1\ b \colon 77\ c \colon 3\ d \colon 4)$.

The **signature of the Oz Universe** consists of literals and integers (constants denoting themselves) and some predicates called **constraint predicates**. The constraint predicates for records are defined as follows:

- $\mathit{label}(x, y)$ holds if and only if x is a record whose label is y.
- $\mathit{width}(x, y)$ holds if and only if x is a record whose width is y.
- $\mathit{subtree}(x, y, z)$ holds if and only if x is a tuple or record, y is a feature of x, and z is the subtree of x at y.
- $\mathit{adjoin}(x, y, z)$ is the predicate corresponding to record adjunction.
- $\mathit{adjoinAt}(x, y, z, u)$ holds if and only if x and u are records such that $x * l(y \colon z) = u$, where l is the label of x.

Constraint predicates for integers are $\mathit{intPlus}(x, y, z)$, $\mathit{intMinus}(x, y, z)$ and $\mathit{intMult}(x, y, z)$ corresponding to the addition, subtraction and multiplication functions on integers.

1.4 An Informal Computation Model

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 [20] for a formal presentation). A full description of Kernel Oz, a semantically complete sublanguage of Oz is given in [6].

1.4.1 The Computation Space

Oz generalizes the model of concurrent constraint programming [15] by providing for higher-order programming and cells. Central to the computation model of Oz is the notion of a **computation space**. A computation space consists of a number of **elaborators** connected to a **blackboard**.

Blackboard

Elaborator $\cdots$ Elaborator

The elaborators read the blackboard and reduce once the blackboard contains sufficient information. The elaborators may reduce in parallel, however the effect must always be achievable by a sequence of single elaborator reductions (interleaving semantics).

The blackboard stores a constraint (constraints are closed under conjunction, hence one constraint suffices) and name bindings. Name bindings map names to abstractions or variables as explained later in this section.

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 $\phi \to \psi$ is valid in the universe, where ϕ is the constraint stored on the blackboard. We say that the 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.

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

1.4.2 Elaboration of Expressions

Elaborators reduce expressions. When an elaborator reduces, it may put new information on the blackboard and create new elaborators. The elaborators of the computation space are short-lived: once they reduce they disappear.

The abstract syntax of expressions is defined as follows:

E	$::=$	ϕ	constraint
	$\mid$	$x\colon \overline{y}/E$	abstraction
	$\mid$	$x\colon y$	cell
	$\mid$	$E\ F$	composition
	$\mid$	$\texttt{local}\ \overline{x}\ \texttt{in}\ E\ \texttt{end}$	declaration
	$\mid$	$x\ y_1 \ldots y_n$	application
	$\mid$	$\texttt{exch}[x, y, z]$	exchange
	$\mid$	$\texttt{if}\ \overline{x}\ \texttt{in}\ \phi\ \texttt{then}\ E\ \texttt{else}\ F\ \texttt{fi}$	conditional
x, y, z	$::=$	⟨*variable*⟩	
$\overline{x}, \overline{y}$	$::=$	⟨*possibly empty sequence of variables*⟩	

By **elaboration** of an expression E we mean the reduction of an elaborator for E. 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 in concurrent constraint programming [15].
- **an abstraction** $x\colon \overline{y}/E$ chooses a fresh name a, binds a to the abstraction $\overline{y}/E$ (name binding) and creates an elaborator for the constraint $x \doteq a$. Since fresh names are chosen whenever a name binding is written on the blackboard, a name cannot be bound to more than one abstraction. Thus elaboration of an abstraction provides it with a unique identity. Since the variable x refers to a name rather than to the abstraction, we can test for equality between x and other variables.
- **a cell** $x\colon y$ chooses a fresh name a, binds a to y (name binding), and creates an elaborator for the constraint $x \doteq a$.
- **a composition** $E\ F$ creates two separate elaborators for E and F.

- **a variable declaration** **local** x **in** E **end** chooses a fresh variable y and an elaborator for the expression $E[y/x]$. The notation $E[y/x]$ stands for the expression that is obtained from E by replacing all free occurrences of x with y. A multiple variable declaration **local** $x\ \overline{x}$ **in** E **end** is treated as a nested declaration **local** x **in** **local** $\overline{x}$ **in** E **end** **end**.
- **an application** $x\ y_1\ \ldots\ y_n$ waits until there is a name a such that the blackboard entails $x \doteq a$. If a is bound to an abstraction $y_1 \ldots y_n/E$, an elaborator for $E[\overline{y}/\overline{z}]$ (a copy of the body of the abstraction, where the actual arguments replace the formal arguments) is created. Otherwise, the application cannot reduce.
- **an exchange** **exch**$[x, y_1, y_2]$ waits until there is a name a such that the blackboard entails $x \doteq a$. If a is bound to a variable z, an elaborator for the constraint $y_1 = z$ is created and the name binding for a is changed such that a is now bound to the variable y_2. Otherwise, the exchange cannot reduce.
- **a conditional** **if** $\overline{x}$ **in** ϕ **then** E **else** F **fi** waits until the blackboard either entails $\exists\overline{x}\ \phi$, in which case an elaborator for the expression **local** $\overline{x}$ **in** ϕ E **end** is created, or disentails $\exists\overline{x}\ \phi$, in which case an elaborator for F is created.

The treatment of abstractions and applications provides for all higher-order programming techniques [21]. By making variables denote names rather than higher-order values, we obtain a smooth combination of first-order constraints with higher-order programming.

While the constraint on the blackboard becomes monotonically stronger over time and bindings of names to abstractions do not change, an exchange may change the binding of a name to a variable. Thus, cells provide a primitive to express state.

1.5 The Programming Language

Having introduced an informal computation model for Oz using the abstract syntax of expressions, we can now present the concrete programming language. In the concrete syntax of Oz, abstractions, applications, cells, exchanges and constraints may not be used directly. Instead the concrete syntax given in Section 1.5.1 must be used for abstraction and application, the concrete syntax given in Section 1.5.2 for cells and exchanges and the concrete syntax given in 1.5.3 for constraints. The execution of a program E amounts to the creation of an elaborator for the expansion of E according to the following sections.

1.5.1 Procedures

In the concrete syntax, variables start with a capital letter to distinguish them from atoms. A procedure `P` taking n arguments can be defined with the concrete syntax

proc {P $X_1 \ldots X_n$} E **end**

standing for the expression

local A **in**
  A:$X_1 \ldots X_n/E$
  P=procedure(\`NAME\`:A \`ARITY\`:n)
end

Thus, a procedure is represented by a record (the concrete syntax for record construction will be explained in Section 1.5.3). This record has the name to which the abstraction is bound as subtree at feature `` `NAME` ``. The variables `` `NAME` `` and `` `ARITY` `` are constrained to names and may not be used in programs.

An application of a procedure `P` to the arguments $X_1, \ldots, X_n$ can be written with the concrete syntax

{P $X_1 \ldots X_n$}

standing for the expression

if A **in**
  label(P,procedure)
  subtree(P,\`NAME\`,A)
  subtree(P,\`ARITY\`,n)
then $A\, X_1 \ldots X_n$
else false fi

Introducing abstractions and applications indirectly in this way enhances programming security in that no application $x\ y_1 \ldots y_n$ may become elaborated unless there exists a name a such that the blackboard entails $x = a$ and a is bound to an n-ary abstraction. If x is constrained to something else but a name, or if the name is not bound to an abstraction or if the arity does not match the arity of the application, the constraint **false** is elaborated, resulting in a run-time error. The effect is a form of dynamic type checking. The representation of procedures by records has additional benefits for objects (see Section 1.6).

1.5.2 Cells

The same form of dynamic type checking as for procedures applies to cells in the concrete syntax of Oz. A cell is created by applying the procedure `NewCell`, defined by

```
proc {NewCell Init C}
   local A in
      A:Init
      C = cell(`NAME`:A)
   end
end
```

and an exchange is performed using the procedure `Exchange` defined as

```
proc {Exchange C X Y}
   if A in
      label(C,cell)
      subtree(C,`NAME`,A)
   then exch[A,X,Y]
   fi
end
```

Note that the default for the missing **else** part of the conditional is **else false**.

1.5.3 Constraints

Because the Oz Universe provides for integers with constraint predicates for addition and multiplication, satisfiability of constraints is undecidable even for conjunctions of atomic integer constraints (Hilbert's Tenth Problem). Therefore, the concrete syntax restricts the use of constraints such that satisfiability and entailment of the occurring constraints is efficiently decidable.

The procedure `Det` plays a key role in the rest of this section. Informally `{Det X}` is entailed whenever `X` becomes determined, i.e. constrained to a record or an integer. The procedure `Det` is defined by

```
proc {Det X}
   if X=1 then true else true fi
end
```

The concrete syntax allows to enter arithmetic constraints like *intPlus*(X, Y, Z) only by expressions of the form `Z=X+Y` which expands to the expression

if `{Det X} {Det Y}` **then** *intPlus*(X, Y, Z) **fi**

This treatment of arithmetic constraints avoids the undecidability problem because the elaboration of a constraint *intPlus*(X, Y, Z) either fails or is equivalent to $\mathsf{Z} = n$ where n is the sum of the integers `X` and `Y`.

We ask the reader to accept a technical inaccuracy here: According to Section 1.4, the guard must consist of a constraint and not contain applications like `{Det X}` (flat guards). Due to space limitations, we will not describe the more complex deep-guard computation here (see [20] for a complete description).

A similar technique as for arithmetic constraints is used to weaken the semantics of record constraints. Instead of using the constraint predicates *label*, *adjoin* and *adjoinAt*, we use the procedures **`Label`**, **`Adjoin`** and **`AdjoinAt`**:

```
proc {Label X Y}
   if {Det X} then label(X,Y) fi
end
proc {Adjoin X Y Z}
   if {Det X} {Det Y} then adjoin(X,Y,Z) fi
end
proc {AdjoinAt X Y Z U}
   if {Det X} {Det Y} then adjoinAt(X,Y,Z,U) fi
end
```

The expression `X.Y=Z` stands for

if `{Det X} {Det Y}` **then** *subtree*(X, Y, Z) **fi**

For record construction, we use the syntax

`X=`$\mathsf{Y}(\mathsf{Y_1}\!:\mathsf{Z_1}\ldots\mathsf{Y_n}\!:\mathsf{Z_n})$

which stands for

if `{Det Y}` `{Det` Y_1`}` $\cdots$ `{Det` Y_n`}`
then
label(X, Y) *width*(X, n) *subtree*$(\mathsf{X}, \mathsf{Y}_1, \mathsf{Z}_1) \ldots$ *subtree*$(\mathsf{X}, \mathsf{Y}_n, \mathsf{Z}_n)$
fi

We write $\mathsf{Y}(\mathsf{Z}_1 \ldots \mathsf{Z}_n)$ as a short hand for $\mathsf{Y}('1'\!:\mathsf{Z}_1 \ldots 'n'\!:\mathsf{Z}_n)$. Thus we obtain Prolog's finite trees as a special case of records. The outlined constraint system is in fact a conservative extension of Prolog II's rational tree system.

1.5.4 Examples

Cells are used to express objects as procedures with state. A simple procedure with state is shown in Program 1.5.1.

Program 1.5.1 A Procedure With State

```
local Cell
in
   {NewCell 0 Cell}
   proc {Num X}
      local Y in {Exchange Cell X Y}  Y = X + 1 end
   end
end
```

Elaboration of this expression creates a local variable `Cell` and an elaborator for the composition. Elaboration of the composition constrains the varaibles `Cell` and `Num` to records and writes two name bindings on the blackboard.

Suppose the computation space contains the applications

```
{Num X} {Num Y} {Num Z}
```

The abstraction realizing the procedure `Num` will be applied concurrently to the variables `X`, `Y`, and `Z`. They will be equated to different numbers and the internal counter of `Num` will be incremented three times. One possible outcome is `X=0 Y=2 Z=1`. The procedure `Num` builds a state sequence

$X_1, X_2, X_3, \ldots, X_k$

whose members are linked by constraints $X_{k+1} = X_k + 1$, and whose respective last member is held in `Cell`. Concurrent applications of `Num` create concurrent exchange requests for `Cell`, which are performed in indeterminate order. Reduction of an application `{Exchange Cell X Y}` will equate `X` to the current end of the sequence and make `Y` the new end of the sequence.

Object-oriented programming in Oz makes use of records to represent states, messages, and method tables. An example for the state of an object is

```
CounterState=state(val:0)
```

The procedure `Inc`

```
proc {Inc State Message Self NewState}
   if Message=inc then {AdjoinAt State val State.val+1
NewState} fi
end
```

increments the value in field `val` of the argument `State`, resulting in `NewState`. We use functional notation in the **then** part which stands for

```
local X Y Z in
   X = val
   Y = State.X
   Z = Y + 1
   {AdjoinAt State X Z NewState}
end
```

The application `{Inc CounterState inc _ NewCounterState}`, where the symbol `_` denotes an anonymous variable occurring only once, constrains `NewCounterState` to `state(val:1)`. The third formal parameter `Self` of the procedure `Inc` is not used in its body, but will serve to capture the notion of "self" in Section 1.6.2 in similar procedures.

Similarly, the procedure `Get`

```
proc {Get State Message Self NewState}
   if X in Message=get(X) then NewState=State
X=State.val fi
end
```

serves to access the value in field `val` and leaves the State unchanged. The application `{Get NewCounterState get(X) _ NewCounterState2}` equates `NewCounterState2` to `NewCounterState` and the variable `X` to `1`.

The variable `CounterMethodTable` in

```
CounterMethodTable=methods(inc: Inc get: Get)
```

is constrained to a record that contains the procedures `Inc` and `Get`. The application of `Inc` above can now be written as

```
{CounterMethodTable.inc CounterState inc _
NewCounterState}
```

1.6 Objects

Our goal are objects with the following properties:

- **Identity and state.** While enjoying persistent indentity, an object changes its behavior over time depending on its state. The manipulation of this state happens in a controlled manner.
- **Structured programming.** The behavior of objects is described in a way that allows code reuse (multiple inheritance, "self").
- **Concurrency.** Objects may be dynamically created and interact with each other in a concurrent setting.

The first goal, we achieve by representing an object by a procedure with state similar to the procedure `Num` in Program 1.5.1. In Section 1.6.1, we refine this scheme by incorporating late method binding and a generic mechanism to create objects.

The second goal is achieved by encoding the behavior of an object by a method table, a record containing methods. Methods are procedures

$method$: $state \times message \times object \rightarrow state$

When the object is applied to a message (represented as a record), the appropriate method is retrieved from the object's method table and applied to the current state, the message, and the object itself, resulting in a new state. We represent method tables by records. In Section 1.6.3, we will show how we can express multiple inheritance by adjunction of method tables, and how the notion of "self" can be captured.

Objects are concurrent due to the inherent concurrency of Oz. In Section 1.6.3, we show how we can nonetheless preserve the order of messages and how objects are synchronized.

1.6.1 Objects Are Procedures With State

Objects are procedures with state whose behavior is determined by a method table. Procedures with state were already discussed in Section 1.5.4.

Program 1.6.1 defines an object `Counter`, employing late method binding. The variable `CounterMethodTable` refers to the record given in Section 1.5.4 on page 12. In the following, we discuss Program 1.6.1 top-down.

Program 1.6.1 A Counter Object

```
local Cell in
   {NewCell state(val:0) Cell}
   proc {Counter Message}
      local State NewState in
         {Exchange Cell State NewState}
         if {Det State}
         then {CounterMethodTable.{Label Message}
                 State Message Counter NewState}
         fi
      end
   end
end
```

The state of the object is represented by a record and stored in `Cell`. The initial content of the cell is the record `state(val:0)`.

When the object `Counter` is applied to a message like `{Counter inc}`, the current `State` is obtained from `Cell` and exchanged with the fresh variable `NewState`. If `State` is determined, the appropriate method `Inc` is retrieved from `CounterMethodTable` using the label `inc` of the message. The method is then applied to `State`, the message `inc`, `Counter` and `NewState`. Thus, if `{Counter inc}` is the first application of `Counter`, `Cell` will hold the new state `state(val:1)`.

Since objects are represented as procedures, they enjoy persistent identity (recall the translation of **proc** $\cdots$ **end** given in Section 1.5.1). Thus one can test for identity of two objects Counter, Counter2 using a conditional **if** Counter = Counter2 **then** $\cdots$ **fi**.

Note that many agents may know the object `Counter` and thus may concurrently attempt to apply `Counter`. Representing the state by a cell ensures mutual exclusion: the respective method applications are implicitly and indeterministically sequentialized.

Generic Object Creation

Since procedures are first-class citizens, we can write a generic procedure shown in Program 1.6.2 that creates a new object `O` from an initial state `IState` and a `MethodTable`.

Program 1.6.2 Generic Object Creation

```
proc {Create IState MethodTable O}
   local Cell in
      {NewCell IState Cell}
      proc {O Message}
         ...
      end
   end
end
```

When `Create` is applied as in

```
{Create state(val:7) CounterMethodTable Counter2}
```

a new counter `Counter2` is created with initial value `7`.

1.6.2 Inheritance

The behavior of an object is determined by its method table. Inheritance thus means that the method table of a new object is obtained by combining and extending method tables of existing objects. Since method tables are represented by records, combining and extending them is straightforward (e.g., by record adjunction). To make the methods of an object accessible, we will now enrich the representation of objects with information used for inheritance. Since objects are procedures and procedures are represented by records on the blackboard, we can construct an enriched object `OInh` by adjoining inheritance information to an object `O`. Program 1.6.3 modifies Program 1.6.2 to incorporate inheritance.

The procedure `Create` now has an additional argument `FromObjects`, a list of objects from which the new object `OInh` inherits. The argument `NewMethodTable` refers to the new methods of the new object. The `MethodTable` is constructed by the procedure `Inherit` by adjoining all method tables of inherited objects and the `NewMethodTable` (we assume the procedure `FoldL` to be known from functional programming).

Program 1.6.3 Incorporating Inheritance

```
proc {Create FromObjects IState NewMethodTable OInh}
   local Cell MethodTable in
      {NewCell IState Cell}
      {Inherit FromObjects NewMethodTable MethodTable}
      proc {O Message} ··· end
      {AdjoinAt O methods MethodTable OInh}
   end
end
```

```
proc {Inherit From NMT MT}
   {Adjoin {FoldL
              From proc {I E O} {Adjoin I E.methods O}
end methods}
    NMT MT}
end
```

Record adjunction (see Section 1.3) takes care of the usual method overriding in object-oriented languages.

`MethodTable` is adjoined to the created procedure `O` to provide the object `OInh` with information that can be used when another object inherits from `OInh`. For example, in Program 1.6.4 a `DecCounter` is created that inherits from `Counter` and additionally understands a message `dec`.

Syntactic Extension

Oz supports a syntactic extension for object creation and method definition, which allows writing the expression in Program 1.6.4 including `CounterMethodTable` in Section 1.5.4 as shown in Program 1.6.5.

The first and the last argument of methods are the incoming `State` and the outgoing `NewState` of the object (see Program 1.6.1). In the body of methods, `NewState` is computed from `State`. During this computation, it may be necessary to introduce several auxiliary state variables. Thus one can say, that the state of the object is threaded through the body of methods. In the syntactic extension, this threading is done by the compiler. The two expressions that implicitly refer to the state are attribute access (`@`) and assignment (`<-`). Syntactic limitations guar-

Program 1.6.4 Example for Inheritance

```
{Create nil state(val:7) CounterMethodTable Counter3}
local DecMethodTable in
   DecMethodTable
      =methods(dec: proc {State Message Self NewState}
                       if Message=dec
                       then {AdjoinAt
                             State val State.val-1
NewState}
                       fi
                    end
              )
   DecCounter={Create Counter3|nil state(val:10)
DecMethodTable}
end
```

Program 1.6.5 Objects in Sugared Syntax

```
create Counter3
   from UrObject
   attr val:7
   meth inc    val <- @val + 1 end
   meth get(X) X = @val        end
end

create DecCounter
   from Counter
   attr val:10
   meth dec val <- @val - 1 end
end
```

antee that there is always only one reference to the state of an object at run-time. Therefore, we can implement assignment such that the construction of a new record as in `AdjoinAt` is avoided (compile-time garbage collection).

Observe that our model alleviates the distinction between classes and their instances by combining object creation and inheritance into one single operation.

Self

The third formal parameter of the methods is the variable `Self`. Since methods are called with the receiving object as third actual parameter, the variable `Self` used in the body of methods has the semantics familiar from object-oriented languages. In the syntactic extension, the keyword **self** represents that variable. For example, the object `Counter4` in Program 1.6.6 sends the message `inc` twice to itself when it receives the message `inc2`.

Program 1.6.6 Example for **self**

```
create Counter4
   from Counter
   attr val:0
   meth inc2 {self inc} {self inc} end
end
```

Method Application

In Section 1.6.2, we saw that attribute access and assignment implicitly refer to the state. In this section, we describe a third such expression, called **method application**.

Assume that the object `Counter4` in Program 1.6.6 has received the message `inc2`. Due to concurrent execution, a message, say `get(X)`, may be received by **self** after the first and before the second `inc` message. To avoid such situations, Oz provides for method application, a way to apply a method locally to the available state. For example, consider

```
create Counter5
   from Counter
   attr val:0
```

```
   meth inc2 <<self inc>> <<self inc>> end
end
```

The state is threaded through the two consecutive method applications introducing an intermediate state

```
local TmpState in State<<self inc>>TmpState
TmpState<<self inc>>NewState end
```

and a threaded method application

```
InState<<O Message>>OutState
```

expands to

```
{O.methods.{Label Message} InState Message Self
OutState}
```

The notation for method application exploits the fact that in our model every method `m` of every object `O` can be referred to by `O.methods.m`. Incidentally, our notation for method application also serves the purpose of Smalltalk's "super" notation. For example, the method `inc` in

```
create Counter6
   from Counter
   attr val:0
   meth inc(X) <<Counter inc>> <<self get(X)>> end
end
```

is defined in terms of `Counter`'s method `inc` and `Counter6`'s own method `get`.

1.6.3 Concurrency Issues

We saw in the previous section that the execution order of applications may not coincide with the textual order of the applications. Using method application, we can define batch methods as a way to enforce an order on messages like in

```
create BatchObject
   meth M|Mr <<self M>> <<self Mr>> end
   meth nil  true end
end
```

The object `Counter4` in Program 1.6.6 can be reformulated using the batch methods.

```
create Counter7
   from Counter BatchObject
   attr val:0
   meth inc2 <<self inc|inc|nil>> end
end
```

Observe the use of multiple inheritance. Sending the message `{Counter7 inc|get(X)|inc|nil}` guarantees that no other application `{Counter7 get(Y)}` can be sent such that `X=Y`.

So far, the application of an object to messages was done in an asynchronous fashion. We can synchronize objects by using messages that are constrained by the object. For example, in

```
{Counter7 inc|get(X)|nil}
if {Det X} then E fi
```

the expression E is only elaborated after `Counter7` is incremented.

1.7 Summary

Oz is an attempt to create a high-level concurrent programming language bringing together the merits of logic and object-oriented programming. For this purpose, we extend the concurrent constraint model with a facility for higher-order programming and the notion of cells. We presented aspects of the underlying constraint system and an informal model of computation of a sublanguage of Oz, based on elaboration of expressions.

We have shown how concurrent objects can be expressed concisely and naturally in Oz. Being represented by named procedures, objects enjoy persistent identity. An object can refer to an encapsulated state, stored in a cell that can only be accessed by calling the object.

Structured programming is supported by late method binding, which is achieved by method lookup in a method table represented by a record. We gave a straightforward implementation of "self" and presented how methods can be applied directly within methods, generalizing the concept of "super". We showed how method tables of several objects may be combined providing for multiple inheritance.

Objects in Oz are concurrent due to the inherent concurrency of Oz. We showed programming techniques that nonetheless enforce an order on messages and allow for synchronization of objects.

Acknowledgements

We thank all members of the Programming Systems Lab at DFKI for inspiring discussions on all kinds of subjects and objects. The research reported in this paper has been supported by the Bundesminister für Forschung und Technologie, contract ITW 9105 (Hydra), and by the ESPRIT basic research project 7195 (ACCLAIM).

Bibliography

[1] F. Benhamou and A. Comerauer, editors. *Constraint Logic Programming*. ISBN0-262-02353-9 III. Series. MIT Press, 1993. Selected Research.

[2] W. Chen, M. Kifer, and D. S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15:187–230, 1993.

[3] K. Clark and S. Gregory. A relational language for parallel programming. In *Proc. of the ACM Conference on Functional Programming Languages and Computer Architecture*, pages 171–178, 1981.

[4] A. Colmerauer, H. Kanoui, and M. V. Caneghem. Prolog, theoretical principles and current trends. *Technology and Science of Informatics*, 2(4):255–292, 1983.

[5] Y. Goldberg, W. Silverman, and E. Shapiro. Logic programs with inheritance. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 951–960, Tokyo, Japan, 1992. ICOT.

[6] 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, dfkiaddr, 1994. Available through anonymous ftp from `ps-ftp.dfki.uni-sb.de` or through www from `http://ps-www.dfki.uni-sb.de`.

[7] 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.

[8] J. Jaffar and M. Maher. Constraint logic programming - a survey. *The Journal of Logic Programming*, 1994. Special issue on 10 years of logic programming.

[9] S. Janson and S. Haridi. Programming paradigms of the Andorra kernel language. In V. Saraswat and K. Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium*, pages 167–186, San Diego, USA, 1991. The MIT Press.

[10] K. Kahn. Objects: A fresh look. In *Proceedings of the Third European Conference on Object Oriented Programming*, pages 207–223. Cambridge University Press, Cambridge, MA, 1989.

[11] 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, Cambridge, MA, 1987. The MIT Press.

[12] R. Milner. Functions as Processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[13] G. Nadathur and D. Miller. An overview of λProlog. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 810–827, Seattle, Wash., 1988. The MIT Press.

[14] 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.

[15] V. A. Saraswat. *Concurrent Constraint Programming*. The MIT Press, Cambridge, Mass., 1993.

[16] C. Schulte and G. Smolka. Encapsulated search in higher-order concurrent constraint programming. In M. Bruynooghe, editor, *Logic Programming: Proceedings of the 1994 International Symposium*, Ithaca, New York, USA, Nov. 1994. The MIT Press.

[17] C. Schulte, G. Smolka, and J. Würtz. Encapsulated search and constraint programming in Oz. In *Second Workshop on Principles and Practice of Constraint Programming*, Orcas Island, Washington, USA, May 1994. Springer-Verlag, LNCS. to appear.

[18] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–511, September 1989.

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

[20] 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 `ps-ftp.dfki.uni-sb.de`.

[21] 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, München, Germany, 7–9 Sept. 1994. Springer-Verlag. Invited Lecture. To appear.

[22] G. Smolka and R. Treinen. Records for logic programming. *The Journal of Logic Programming*, 18(3):229–258, Apr. 1994.