The Oz Programming Model

Gert Smolka
Programming Systems Lab
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany
email: `smolka@dfki.uni-sb.de`

July 1995

Abstract

The Oz Programming Model (OPM) is a concurrent programming model subsuming higher-order functional and object-oriented programming as facets of a general model. This is particularly interesting for concurrent object-oriented programming, for which no comprehensive formal model existed until now. The model can be extended so that it can express encapsulated problem solvers generalizing the problem solving capabilities of constraint logic programming. OPM has been developed together with a concomitant programming language Oz, which is designed for applications that require complex symbolic computations, organization into multiple agents, and soft real-time control. An efficient, robust, and interactive implementation of Oz is freely available.

This paper has appeared in: Computer Science Today, Jan van Leeuwen, editor, Lecture Notes in Computer Science, Volume 1000, pages 324–343, Springer-Verlag, Berlin, 1995.

Contents

1 Introduction 3

2 Computation Spaces 4

3 Concurrency and Parallelism 5

4 Synchronization as Logic Entailment 5

5 Constraint Structures 6

6 A Simple Concurrent Constraint Language 7

7 First-class Procedures 9

8 Cells 11

9 Ports 12

10 Names 13

11 Agents 13

12 Objects 15

13 Distribution 16

14 Incremental Tell 16

15 Propagators 18

16 Threads 18

17 Time 19

18 Encapsulated Search 20

19 Summary 20

1 Introduction

Computer systems are undergoing a revolution. Twenty years ago, they were centralized, isolated, and expensive. Today, they are parallel, distributed, networked, and inexpensive. However, advances in software construction have failed to keep pace with advances in hardware. To a large extent, this is a consequence of the fact that current programming languages were conceived for sequential and centralized programming.

A basic problem with existing programming languages is that they delegate the creation and coordination of concurrent computational activities to the underlying operating system and network protocols. This has the severe disadvantage that the data abstractions of the programming language cannot be shared between communicating computational agents. Thus the benefits of existing programming languages do not extend to the central concerns of concurrent and distributed software systems.

Given this state of affairs, the development of concurrent programming models is an important research issue in Computer Science. A concurrent programming model must support the creation and coordination of multiple computational activities. Simple concurrent programming models can be obtained by accommodating concurrency in the basic control structure of the model. This way concurrency appears as a generalization rather than an additional feature.

The development of simple, practical, high-level, and well-founded concurrent programming models turned out to be difficult. The main problem was the lack of a methodology and formal machinery for designing and defining such models. In the 1980's, significant progress has been made on this issue. This includes the development of abstract syntax and structural operational semantics [17, 27]; functional and logic programming, two declarative programming models building on the work of logicians (lambda calculus and predicate logic); CCS [12] and the π-calculus [13], two well-founded concurrent programming models developed by Milner and others; and the concurrent constraint model [10, 18], a concurrent programming model that originated from application-driven research in concurrent logic programming [21] and constraint logic programming [6].

This paper reports on the Oz Programming Model, OPM for short, which has been developed together with the concurrent high-level programming language Oz. OPM is an extension of the basic concurrent constraint model, adding first-class procedures and stateful data structures. OPM is a concurrent programming model that subsumes higher-order functional and object-oriented programming as facets of a general model. This is particularly interesting for concurrent object-oriented programming, for which no comprehensive formal model existed until now. There is a conservative extension of OPM providing the problem-solving capabilities of constraint logic programming. The resulting problem solvers appear as concurrent agents encapsulating search and speculative computation with constraints.

Oz and OPM have been developed at the DFKI since 1991. Oz [25, 23, 22] is designed

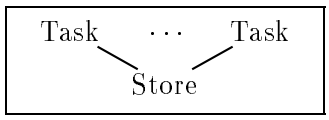
as a concurrent high-level language that can replace sequential high-level languages such as Lisp, Prolog and Smalltalk. There is no other concurrent language combining a rich object system with advanced features for symbolic processing and problem solving. First applications of Oz include simulations, multi-agent systems, natural language processing, virtual reality, graphical user interfaces, scheduling, time tabling, placement problems, and configuration. The design and implementation of Oz took ideas from AKL [7], the first concurrent constraint language with encapsulated search.

An efficient, robust, and interactive implementation of Oz, DFKI Oz, is freely available for many Unix-based platforms (see remark at the end of this paper). DFKI Oz features a programming interface based on GNU Emacs, a concurrent browser, an object-oriented interface to Tcl/Tk for building graphical user interfaces, powerful interoperability features, an incremental compiler, and a run-time system with an emulator and a garbage collector.

DFKI Oz proves that an inherently concurrent language can be implemented efficiently on sequential hardware. Research on a portable parallel implementation for shared memory machines has started. More ambitiously, we have also begun work towards a distributed version of Oz supporting the construction of open systems.

This paper describes OPM in an informal manner. Calculi formalizing the major aspects of OPM can be found in [23, 22]. The Oz Primer [25] is an introduction to programming in Oz. Basic implementation techniques for Oz are reported in [11].

2 Computation Spaces



Computation in OPM takes place in a *computation space* hosting a number of *tasks* connected to a shared *store*. Computation advances by *reduction of tasks*. The reduction of a task can manipulate the store and create new tasks. When a task is reduced it disappears. Reduction of tasks is an atomic operation, and tasks are reduced one by one. Thus there is no parallelism at the abstraction level of OPM.

Tasks can *synchronize* on the store in that they become reducible only once the store satisfies certain conditions. A key property of OPM is that *task synchronization is monotonic*, that is, a reducible task stays reducible if other tasks are reduced before it.

Typically, many tasks are reducible in a given state of a computation space. To obtain fairness, reactivity, and efficiency, a *reduction strategy* is needed to select the reducible tasks qualifying for the next reduction step. *Fairness* ensures that several groups of tasks can advance simultaneously. *Reactivity* means that one can create computations that react to outside events within foreseeable time bounds. The following is an example of a fair and reactive reduction strategy:

> All tasks are maintained in a queue, where the first task of the queue is the one to be considered next for reduction. If it is not reducible, it is moved to the

end of the queue. If it is reducible, it is reduced and the newly created tasks are appended at the end of the queue.

We will see later that this strategy is inefficient since its degree of fairness is too fine-grained for OPM. A practical reduction strategy will be given in Section 16.

3 Concurrency and Parallelism

OPM is a concurrent and nonparallel programming model. Concurrency means that one can create several simultaneously advancing computations, possibly synchronizing and communicating. Parallelism means that the execution of several hardware operations overlaps in time. Concurrency can be obtained in a nonparallel setting by interleaving reduction steps. This is typically the case in operating systems that advance several concurrent processes on single processor machines. We can see concurrency as a programming abstraction and parallelism as a physical phenomenon.

The fact that OPM is nonparallel does not exclude a parallel implementation, however. The reason for making OPM concurrent but not parallel is the desire to make things as simple as possible for programmers. In OPM, the semantics of programs does not depend on whether they run on a sequential or parallel implementation. Thus the complexities of parallelism need only concern the implementors of OPM, not the programmers.

4 Synchronization as Logic Entailment

We will now see how OPM realizes monotonic task synchronization. The basic idea is very simple. We assume that a set of logic formulas, called *constraints*, is given. The set of constraints is closed under conjunction, and for constraints a logic entailment relation ("C implies D") is defined. We also assume that the store of a computation space holds a constraint in a special compartment, called the *constraint store*. The only way the constraint store can be updated is by *telling* it a constraint C, which means that the constraint store advances from S to the conjunction $S \wedge C$. Finally, we assume that it is possible to synchronize a task on a constraint, called its *guard*. A synchronized task becomes reducible if its guard is entailed by the constraint store.

It is easy to see that this synchronization mechanism is monotonic. At any point in time, the constraint store can be seen as a conjunction

$$\texttt{true} \wedge C_1 \wedge C_2 \wedge \ldots \wedge C_n$$

where $C_1, \ldots, C_n$ are the constraints told so far. The beauty of this arrangement is that the information in the constraint store increases monotonically with every further constraint told, and that the order in which constraints are told is insignificant as far as the information in the store is concerned (conjunction is an associative and commutative operation).

We assume that the constraint store is always satisfiable. Consequently, it is impossible to tell a constraint store S a constraint C if the conjunction $S \wedge C$ is unsatisfiable.

It suffices to represent the constraint store modulo logic equivalence. This means that the synchronization mechanism is completely declarative. It turns out that there are constraint systems for which synchronization as entailment is both expressive and efficient.

Synchronization on a constraint store appeared first in Prolog II [4] in the primitive form of the so-called freeze construct. The idea to synchronize on entailment of constraints is due to Maher [10].

5 Constraint Structures

We now make precise the notions of constraint and entailment. We will also see that the constraint store is the place where information about the values participating in a computation is stored. An important property of the constraint store is the fact that it can store partial (i.e., incomplete) information about the values of variables.

A *constraint structure* is a structure of first-order predicate logic. The elements of a constraint structure are called *values*, and the first-order formulas over the signature of a constraint structure are called *constraints*. We assume that constraints are built over a fixed infinite alphabet of variables. A constraint C *entails* a constraint D if the implication $C \rightarrow D$ is valid in the constraint structure. A constraint C *disentails* a constraint D if C entails $\neg D$. Two constraints C and D are *equivalent* if C entails D and D entails C.

The constraint structure must be chosen such that its elements are the values we want to compute with. The values will typically include numbers, ordered pairs of values, and additional primitive entities called names. Values can be thought of as stateless data structures. Note that this set-up requires that values are defined as mathematical entities, and that operations on values are described as mathematical functions and relations.

To ensure that checking entailment between the constraint store and guards is computationally inexpensive, one must carefully restrict the constraints that can be written in the constraint store and that can be used as guards.

We now outline a concrete constraint structure INP. As values of INP we take the integers, an infinite set of primitive entities called *names*, and all ordered pairs that can be obtained over integers and names. We write $v_1|v_2$ for the ordered pair whose left component is the value v_1 and whose right component is the value v_2. Moreover, we assume that the signature of INP provides the following *primitive constraints*:

- $x = n$ says that the value of the variable x is the integer n.
- $x = \xi$ says that the value of the variable x is the name ξ.
- $x = y|z$ says that the value of the variable x is the pair having the value of the variable y as left and the value of the variable z as right component.

- $x = y$ says that the variables x and y have the same value.

An example of a constraint store over INP is

$$x = y \ \wedge \ y = z|u \ \wedge \ z = 3.$$

This constraint store asserts that the value of z is 3, that the value of y is a pair whose left component is 3, and that x and y have the same value. While this constraint store has total information about the value of the variable z, it has only partial information about the values of the other variables. In fact, it has no information about any variable other than x, y and z.

The constraint store above entails the constraint $x = 3|u$ and disentails the constraint $x = 3$. It neither entails nor disentails the constraint $y = 3|5$.

In practice, one uses more expressive constraint structures than INP. The constraint structure CFT [26, 2] offers constraints over possibly infinite records called feature trees. Oz employs an extension of CFT.

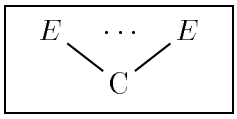
6 A Simple Concurrent Constraint Language

We now present a sublanguage OCC of OPM that is also a sublanguage of Saraswat's concurrent constraint model [18]. OCC cannot yet express indeterministic choice, which we will accommodate later (see Section 8).

The store of an OCC computation space consists only of the constraint store. As constraint structure we take INP to be concrete. As tasks we take expressions according to the abstract syntax

E	$::=$	C	*constraint*
	$\mid$	$E_1 \wedge E_2$	*composition*
	$\mid$	**if** C **then** E_1 **else** E_2	*conditional*
	$\mid$	**local** x **in** E	*declaration*

where C ranges over a suitably restricted class of constraints, and where x ranges over the variables used in constraints. A declaration **local** x **in** E binds the variable x with scope E. *Free* and *bound* variables of expressions are defined accordingly.



An OCC computation space consists of tasks which are expressions as defined above and a store which is a satisfiable constraint. Tasks which are constraints, compositions or declarations are unsynchronized. Conditional tasks synchronize on the constraint store and become reducible only once their guard is entailed or disentailed by the constraint store.

The reduction of a constraint task C *tells* the constraint store the constraint C. We say that such a reduction performs a *tell operation*. If the conjunction $S \wedge C$ of the present constraint

store S and C is satisfiable, the reduction of the task C will advance the constraint store to $S \wedge C$. If the conjunction $S \wedge C$ of the present constraint store S and C is unsatisfiable, the reduction of the task C will not change the constraint store and announce *failure*. A concrete language has three possibilities to handle the announcement of failure: to ignore it, to abort computation, or to handle it by an exception handling mechanism.

Reduction of a composition $E_1 \wedge E_2$ creates two tasks E_1 and E_2. Reduction of a conditional **if** C **then** E_1 **else** E_2 creates the task E_1 if C is entailed and the task E_2 if C is disentailed by the constraint store. Reduction of a declaration **local** x **in** E chooses a fresh variable y and creates the task $E[y/x]$ obtained from E by replacing all free occurrences of x with y. A variable is *fresh* if it does not occur in the current state of the computation space.

The expressions of OCC provide basic operations for concurrent programming. Compositions make it possible to obtain several concurrent tasks from a single task. Conditionals make it possible to synchronize tasks on the constraint store. Telling constraints makes it possible to fire synchronized tasks. Declarations make it possible to obtain fresh variables. This will become significant as soon as we introduce procedures. For now observe that two identical tasks **local** x **in** E will reduce to two different tasks $E[y/x]$ and $E[z/x]$, where y and z are distinct fresh variables.

Telling constraints makes it possible to assert information about the values of variables (e.g., $x = 7$). The combination of conditionals and telling makes it possible to access the constituents of nonprimitive values. The task

$$\textbf{if } \exists y \exists z (x = y|z) \textbf{ then } x = u|v \textbf{ else } E$$

will equate the variables u and v to the left and right component of x if x turns out to be pair, and reduce to the task E otherwise. We call this construction a *synchronized decomposition*. To have a convenient notation, we will write

$$\textbf{if } x_1 \ldots x_n \textbf{ in } C \textbf{ then } E_1 \textbf{ else } E_2$$

as an abbreviation for

$$\textbf{if } \exists x_1 \ \ldots \ \exists x_n \, C \textbf{ then local } x_1 \textbf{ in } \ldots \textbf{ local } x_n \textbf{ in } (C \wedge E_1) \textbf{ else } E_2$$

With that we can write the above task as

$$\textbf{if } y \ z \textbf{ in } x = y|z \textbf{ then } u = y \wedge v = z \textbf{ else } E$$

The reason for having the conditional synchronize symmetrically on entailment and disentailment is that the incremental algorithms for checking entailment automatically also check for disentailment [1, 26]. These algorithms have in fact three outcomes: entailed, disentailed, or neither. The symmetric form of the conditional also has the nice property that it makes negated guards unnecessary since **if** $\neg C$ **then** E_1 **else** E_2 is equivalent to **if** C **then** E_2 **else** E_1.

Given a state of a computation space, we say that a variable x is *bound to* an integer n [a name ξ, a pair] if the constraint store entails the constraint $x = n$ [$x = \xi$, $\exists y \exists z (x = y|z)$].

7 First-class Procedures

Every programming language has procedures. Procedures are the basic mechanism for expressing programming abstractions. If provided in full generality, procedures have spectacular expressivity. As is well-known from the lambda calculus, creation and application of nonrecursive functional procedures alone can express all computable functions.

A programming language provides *first-class procedures* if

- procedures can create new procedures,
- procedures can have lexically scoped global variables,
- procedures are referred to by first-class values.

First-class procedures are available in functional programming languages such as Scheme, SML or Haskell. They are typically not available in today's concurrent programming languages although they can provide crucial functionality for concurrent and distributed programming (see the later sections of this paper and also [3]).

In OPM, a *procedure* is a triple

$$\xi\colon z/E$$

consisting of a name ξ (see Section 5), a *formal argument* z (a variable), and a *body* E (an expression). A procedure binds its formal argument z with scope E. The *free* or *global* variables of a procedure are defined accordingly. Procedures can actually have any number of formal arguments, but for now we consider only one argument to ease our presentation.

Besides the constraint store, OPM's store has a second compartment called the *procedure store*. The procedure store contains finitely many procedures such that for one name there is at most one procedure. Once a procedure has been entered into the procedure store, it cannot be retracted. Information about the values of the global variables of a procedure is kept in the constraint store. What we call a procedure is often called a closure in the literature.

There are two new expressions for creating and applying procedures:

$$\begin{array}{rcll} E & ::= & \textbf{proc}\ \{x\ z\}\ E & \textit{definition} \\ & | & \{x\ y\} & \textit{application} \end{array}$$

A definition **proc** $\{x\ z\}\ E$ binds its *formal argument* z (a variable) with scope E. Definitions are always reducible. The reduction of a definition **proc** $\{x\ z\}\ E$ chooses a fresh name ξ, tells the constraint store the constraint $x = \xi$, and writes the new procedure $\xi\colon z/E$ into the procedure store.

An application $\{x\ y\}$ must wait until the procedure store contains a procedure $\xi\colon z/E$ such that the constraint store entails $x = \xi$. If this is the case, the application task $\{x\ y\}$ can

reduce to the task $E[y/z]$, which is obtained from the body of the procedure by replacing all free occurrences of the formal argument z with the actual argument y, avoiding capturing.

The sublanguage of OPM introduced so far can express both eager and lazy higher-order functional programming [23]. For instance, a higher-order function

$$\mathit{MkMap}\colon (\mathit{Value} \to \mathit{Value}) \;\to\; (\mathit{List} \to \mathit{List})$$

returning a list mapping function can be expressed as a binary procedure

```
proc {MkMap F Map}
   proc {Map Xs Ys}
      if X Xr in Xs=X|Xr then
         local Y Yr in Ys=Y|Yr  {F X Y}  {Map Xr Yr} end
      else Ys=Nil fi
   end
end
```

We are now using concrete Oz syntax, where a composition $E_1 \wedge E_2$ is written as a juxtaposition $E_1\ E_2$. A list $v_1, \ldots, v_n$ is represented as a nested pair $(v_1|(\ldots(v_n|\nu))\ldots)$, where ν is a name representing the empty list. We assume that the variable `Nil` is bound to ν. The procedure `MkMap` takes a binary procedure `F` as input and creates a binary procedure `Map` mapping lists elementwise according to `F`.

Since our model employs logic variables, there is no static distinction between input and output arguments. The functionality offered by a procedure $\xi\colon z/E$ is simply the ability to spawn any number of tasks $E[y/z]$, where the variable y replacing the formal argument z can be chosen freely each time.

To ease our notation, we will suppress auxiliary variables by means of *nesting*. For instance, we will write

```
{{MkMap F} 1|2|Nil X}
```

as an abbreviation for

```
local Map One Two A B in
   {MkMap F Map}  One=1  Two=2  A=One|B  B=Two|Nil  {Map A X}
end
```

The procedure `MkMap` actually implements a *concurrent function*. For instance, the task

```
{{MkMap F} A|B|C X}
```

will tell the constraint `X=U|V|W`, where `U`, `V`, and `W` are fresh variables. It will also create tasks that automatically synchronize on the variables `F`, `A`, `B`, and `C` and that will compute the values of `U`, `V`, and `W` when the necessary information is available.

The representation of functional computation as concurrent computation has been studied carefully for calculi formalizing the relevant aspects of OPM [23, 16, 15]. The main results include the identification of confluent subcalculi, embeddings of the eager and the

lazy lambda calculus, and a correctness proof for the eager embedding. Lazy functional programming can be embedded such that argument computations are shared, a crucial feature of implementations that cannot be modeled with the lambda calculus [9].

OPM combines higher-order programming with first-order constraints. The idea to interface variables and procedures through freshly chosen names appeared first in Fresh [24].

8 Cells

Besides the constraint and the procedure store, OPM's store has a third and final compartment called the *cell store*. A cell is a mutable binding of a name to a variable. Cells make it possible to express stateful and concurrent data structures, which can serve as a communication medium between concurrent agents. There is an exchange operation on cells that combines reading and writing into a single atomic operation, thus providing mutual exclusion and indeterminism as needed for many-to-one communication.

The cell store contains finitely many *cells* $\xi{:}\,x$ representing mutable bindings of names to variables. Similar to the procedure store, the cell store contains at most one cell per name. Given a cell $\xi{:}\,x$ in the cell store, we say that the cell ξ *hosts* the variable x. The task

```
{NewCell X Y}
```

chooses a fresh name ξ, tells the constraint store the constraint $\mathtt{Y} = \xi$, and writes the new cell $\xi{:}\,\mathtt{X}$ into the cell store. Once a cell has been entered into the cell store, it cannot be retracted. The task

```
{Exchange X Y Z}
```

must wait until the cell store contains a cell $\xi{:}\,u$ such that the constraint store entails $\mathtt{X} = \xi$. The task can then be reduced by updating the cell to host the variable `Z` and telling the constraint store the constraint $\mathtt{Y} = u$.

Cells introduce indeterminism into OPM since the order in which multiple exchange tasks for the same cell are reduced is unspecified.

Cells are different from assignable variables in multi-threaded imperative languages. For one thing, OPM ensures mutual exclusion for concurrent exchange tasks for the same cell (since OPM is nonparallel and task reduction is an atomic operation). Moreover, an exchange task combines reading and writing of a cell into a single atomic operation. In the presence of logic variables, this atomic combination turns out to be expressive since one can write a new variable into a cell whose value will be computed only afterwards from the value of the old variable in the cell. This cannot be obtained in an imperative setting since it requires that consumers of a variable are automatically synchronized on the event that the value of the variable becomes known.

9 Ports

Building on cells, we can express complex concurrent data structures with state. The internal structure of such data structures can be hidden by means of procedural abstraction and lexical scoping of variables. We can thus obtain abstract concurrent data types with state.

As a first example we consider ports [8], which can serve as message queues for agents. A *port* is a procedure connected to a stream. A *stream* is a variable `S` that is incrementally constrained to a list by telling a constraint for every element of the list:

$$S=X_1|S_1,\ S_1=X_2|S_2,\ S_2=X_3|S_3,\ S_3=X_4|S_4,\ \ldots$$

It is assumed that nobody but the procedure `P` writes on the stream. An application `{P X}` will tell a constraint $S_i=X|S_{i+1}$, where S_i is the current tail of the stream and S_{i+1} is a new variable serving as the new tail of the stream. A port has state because it must remember the current tail of its stream. A port is a concurrent data structure since it allows several concurrent computations to write consistently on a single stream.

The procedure

```
proc {NewPort Stream Port}
   local Cell in
      {NewCell Stream Cell}
      proc {Port Message}
         local Old New in
            {Exchange Cell Old New}  Old=Message|New
         end
      end
   end
end
```

creates a new port `Port` connected to a stream `Stream`. The port holds the current tail of its stream in a private cell `Cell`. Note how lexical scoping ensures that no one but the port can see the cell. Also note that `NewPort` is a higher-order procedure in that it creates and returns a new procedure `Port`.

How can we enter two messages `A` and `B` to a port such that `A` appears before `B` on the associated stream? To make things more interesting, we are looking for a solution making it possible that other concurrently sent messages can be received between `A` and `B` (it may take a long time before `B` is sent).

One possible solution makes assumptions about the employed reduction strategy (see Section 16). Here we will give a solution that will work for every reduction strategy. The basic idea is to model a port as a binary procedure

```
{Port Message Continuation}
```

that will tell the constraint `Continuation=Port` after `Message` has been put on the stream [8]. Two messages `A` and `B` can then be sequentialized by writing

```
local Continuation Dummy in
   {Port A Continuation} {Continuation B Dummy}
end
```

Such synchronizing ports can be created with

```
proc {NewSyncPort Stream Port}
   local Cell in
      {NewCell Port|Stream Cell}
      proc {Port Message Continuation}
         local New in
            {Exchange Cell Continuation|Message|New Port|New}
         end
      end
   end
end
```

10 Names

Names serve as dynamically created capabilities that cannot be faked. It is often useful to be able to obtain reference to fresh names that do not designate procedures or cells. For this purpose we introduce a primitive task

```
{NewName X}
```

which can be reduced by choosing a fresh name ξ and telling the constraint `X=`ξ. Referring to names by means of variables has the advantage of lexical scoping and also avoids the need for a concrete syntax for names. Using names, lexical scoping, and procedures, sophisticated access control schemes can be expressed.

11 Agents

An agent is a computational abstraction processing messages received through a port. It maintains an internal state and may send messages to other agents. An example of an agent is a queue that can handle concurrent enqueue and dequeue requests.

We assume that the functionality of an agent is given by a procedure

$$\mathit{Serve}\colon\ \mathit{State} \times \mathit{Message} \ \rightarrow\ \mathit{NewState}$$

describing how the agent serves a message and how it advances its state. The procedure

```
proc {NewAgent Serve Init Port}
   local Stream Feed in
      {NewPort Stream Port}
      {Feed Stream Init}
      proc {Feed Ms State}
         if Message Mr NewState in Ms=Message|Mr then
            {Serve State Message NewState}  {Feed Mr NewState}
         else true fi
      end
   end
end
```

creates a new agent that receives messages through `Port` and operates as specified by the procedure `Serve` and the initial state `Init`. Note that an agent hides the stream queueing its messages.

A queue agent receiving messages through a port `Q` can be created with

```
local Xs in {NewAgent QueueServe Xs|Xs Q} end
```

where the procedure `QueueServe` is defined as follows:

```
{NewName Enqueue}
{NewName Dequeue}
proc {QueueServe State Message NewState}
   if First Last in State=First|Last then
      if X NewLast in Message=Enqueue|X then
         Last=X|NewLast  NewState=First|NewLast
      else
         if X NewFirst in Message=Dequeue|X then
            First=X|NewFirst  NewState=NewFirst|Last
         else true fi
      fi
   else true fi
end
```

Messages are represented as pairs `Enqueue|X` and `Dequeue|X`, where the variables `Enqueue` and `Dequeue` are bound to names identifying the corresponding operations. Using lexical scoping, one can construct contexts in which none or only one of the two operations is visible.

A message `Enqueue|X` will enqueue `X`, and a message `Dequeue|X` will dequeue an item and bind it to `X`. In case the queue is empty, a dequeue request will wait in a queue of unserved dequeue requests, which is served as soon as an item is entered into the queue. The procedure `QueueServe` shows that this synchronization idea can be expressed elegantly by means of logic variables.

12 Objects

Objects are a modular programming abstraction for concurrent data structures with state. We model objects as procedures `{Object Message}` that are applied to messages. A message is a pair `MethodName|Argument`. When an object is applied to a message, it invokes the requested method with the given argument and advances to a new state. Similar to agents, we assume that the functionality of an object is specified by a procedure

$$\textit{Serve}\colon\ \textit{State} \times \textit{Message} \times \textit{Self} \ \rightarrow\ \textit{NewState}$$

describing how the agent serves a message and how it advances its state. The argument *Self* is a reference to the object invoking *Serve* making it possible to have a self reference within *Serve* and still share *Serve* between several objects. The procedure

```
proc {NewObject Serve Init Object}
   local Cell in
      {NewCell Init Cell}
      proc {Object Message}
         local State NewState in
            {Exchange Cell State NewState}
            {Serve State Message Object NewState}
         end
      end
   end
end
```

creates a new object `Object` from a procedure `Serve` and an initial state `Init`.

It is straightforward to express classes defining serve procedures in a modular fashion by means of named methods. Methods are modeled as procedures similar to serve procedures. Objects can then be obtained as instances of classes. The states of objects are modeled as finite mappings from attributes to variables, where attributes are modeled as names. Methods can then construct new states from given states by "assigning" variables to attributes. One can also provide for *inheritance*, that is, the ability to construct new classes by inheriting methods and attributes from existing classes. All this is a matter of straightforward higher-order programming. Exploiting the power of lexical scoping and names, it is straightforward to express private attributes and methods.

OPM is a simple and powerful base for expressing concurrent object-oriented programming abstractions. It was in fact designed for this purpose. Concrete programming languages will of course sweeten frequently used programming abstractions with a convenient notation. For a concrete system of object-oriented abstractions and notations we refer the reader to the Oz object system [5, 25].

The reader will have noticed the similarity between agents and objects. We can see agents as active objects. An object can easily be turned into an agent by interfacing it through a port.

13 Distribution

OPM can be extended to serve as a model for distributed programming. Distribution means that a program can spread computations over a network of computers. At the abstraction level of OPM, this can be modeled by assigning a *site* to every task and by assuming that the store is distributed transparently. Moreover, we assume that new tasks inherit the site of the creating task.

We can now see a clear difference between agents and objects. When we send a message to an agent, the message is served at the site where the agent was created (there is a task waiting for the next message sent). When we apply an object to a message, the message is served at the site where the object is applied. In other words, agents are stationary and objects are mobile.

Since OPM has first-class procedures, it is straightforward to express compute servers. Cardelli [3] gives an excellent exposition of distributed programming techniques available in a lexically-scoped language with first-class procedures and concurrent state.

The assumption of a transparently distributed store is not realistic for many applications. It conflicts with the ability to model fault-tolerance, for instance. We have started work on a less abstract model where the store appears as a directed graph whose nodes are situated similar to tasks.

14 Incremental Tell

The tell operation of OCC (see Section 6) is not suitable for a parallel implementation. The reason is that a constraint must be told in a single reduction step. Since telling a constraint (e.g., $x = y$) may involve scanning the entire store, other tell tasks may be blocked for a long time. The problem can be resolved by telling a constraint piecewise. The basic idea is to reduce a constraint task T by keeping the task T as is and by advancing the constraint store from S to a slightly stronger constraint store S' entailed by $S \wedge T$. This amplifying reduction step is repeated until the constraint store entails T, in which case the task T is discarded. Since the constraint store must always be satisfiable, the case where $S \wedge T$ is unsatisfiable needs special care.

To make the incremental tell operation precise, we introduce the notion of a constraint system. A *constraint system* consists of a constraint structure, a set of constraints called *basic constraints*, and, for every basic constraint T, a binary relation $\rightarrow_T$ on basic constraints such that:

1. The basic constraints are closed under conjunction and contain $\perp$ (i.e., false).
2. For every basic constraint T, the relation $\rightarrow_T$ is well-founded, that is, there exists no infinite chain $S_1 \rightarrow_T S_2 \rightarrow_T S_3 \rightarrow_T \cdots$.

3. If $S \rightarrow_T S'$, then (i) S' entails S, (ii) $S \wedge T$ entails S', (iii) S is satisfiable, and (iv) S' is unsatisfiable if and only if $S' = \bot$.

4. If T is not entailed by S and both are basic constraints, then there exists S' such that $S \rightarrow_T S'$.

The *tell reductions* $\rightarrow_T$ correspond to the visible simplification steps of the incremental algorithms implementing the necessary operations on constraint stores. Such algorithms can be found, for instance, in [1, 26]. Note that the tell reductions may be nondeterministic; that is, for given S and T, there may be different S_1 and S_2 such that $S \rightarrow_T S_1$ and $S \rightarrow_T S_2$.

Let S be a satisfiable basic constraint and T a basic constraint. Then the tell reduction $\rightarrow_T$ satisfies the following properties:

1. S entails T if and only if S is irreducible with respect to $\rightarrow_T$.

2. S disentails T if and only if every maximal chain $S \rightarrow_T \cdots$ ends with $\bot$.

3. Let $S \rightarrow_T \cdots \rightarrow_T S'$ be a chain such that S' is irreducible with respect to T. Then (i) $S \wedge T$ is equivalent to S' and (ii) $S \wedge T$ is unsatisfiable if and only if $S' = \bot$.

Given a constraint system, we assume that the constraints appearing in expressions and the constraint store are all basic, where the constraints appearing in guards may be existentially quantified. Given a constraint store S and a constraint task T, the *incremental tell operation* is defined as follows: if S is irreducible with respect to $\rightarrow_T$, then the task T is discarded. Otherwise, choose some basic constraint S' such that $S \rightarrow_T S'$. If $S' = \bot$, then announce failure and discard the task T; if $S' \neq \bot$, then advance the constraint store to S' and keep the task T.

The canonical constraint system for the constraint structure INP comes with the basic constraints

$$C \ ::= \ \bot \mid \top \mid \langle \textit{primitive constraint} \rangle \mid C_1 \wedge C_2.$$

Primitive constraints were defined in Section 5.

As long as failure does not occur, it is not important to know which tell reductions are used. However, if $S \wedge T$ is unsatisfiable and computation can continue after failure (e.g., since there is exception handling), all chains $S \rightarrow_T \cdots \rightarrow_T S'$ should only add local information. The notion of "local information" cannot be made precise in general. However, there are straightforward definitions for INP and other practically relevant constraint systems. Here we will just give an example for INP. Given $S \equiv (x = 1|2 \ \wedge \ y = u|3)$ and $T \equiv (x = y)$, the tell reduction $\rightarrow_T$ should only permit two maximal chains issuing from S: $S \ \rightarrow_T \ S \wedge u = 1 \ \rightarrow_T \ \bot$ and $S \ \rightarrow_T \ \bot$.

15 Propagators

The algorithms for telling and checking entailment and disentailment of basic constraints must be efficient. The typical complexity should be constant time, and the worst-case complexity should be quadratic or better in the size of the guard and the constraint store. Consequently, expressive constraints such as $x + y = z$ and $x * y = z$ cannot be written into the constraint store and hence cannot be accommodated as basic constraints. (For nonlinear constraints over integers satisfiability is undecidable (Hilbert's Tenth Problem).)

Nonbasic constraints can be accommodated as tasks that wait until the constraint store contains enough information so that they can be equivalently replaced with basic constraints. For instance, a task $x + y = z$ may wait until there exist two integers n and m such that the constraint store entails $x = n \wedge y = m$. If this is the case, the task can be reduced to the basic constraint $z = k$, where k is the sum of n and m. Nonbasic constraints that are accommodated in this way are called *propagators*.

Another example of a propagator is a Boolean order test for integers:

$$less(x, y, z) \equiv (x < y \leftrightarrow z = \texttt{True}) \wedge (z = \texttt{True} \vee z = \texttt{False}).$$

`True` and `False` are variables bound to distinct names. This propagator can reduce to $z = \texttt{True}$ or $z = \texttt{False}$ as soon as the constraint store contains sufficient information about the values of x and y.

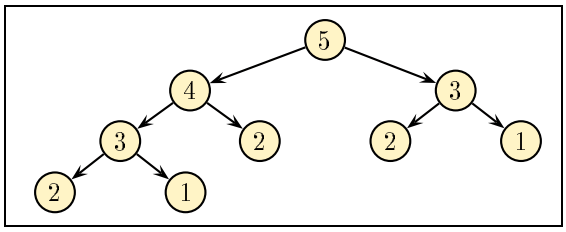
16 Threads

We now give an efficient reduction strategy for OPM that is fair and reactive (see Section 2). An efficient reduction strategy must make it possible to write programs that create only a moderate amount of concurrency, which can be implemented efficiently on both single and multi-processor architectures.

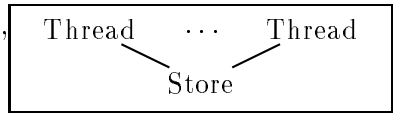
The example of the Fibonacci function (we use a sugared notation suppressing auxiliary variables)

```
proc {Fib N M}
   local B X Y in
      {Less 2 N B}
      if B=True then {Fib N-1 X}  {Fib N-2 Y}  X+Y=M else M=1 fi
   end
end
```

shows that the naive reduction strategy in Section 2 is impractical: it will traverse the recursion tree of `{Fib 5 M}`, say, in breadth-first manner, thus requiring exponential space. On the other hand, sequential execution will traverse the recursion tree in depth-first manner from left to right and will thus only need linear space. This difference clearly matters.



The efficient reduction strategy organizes tasks into threads, where every thread is guaranteed to make progress. Thus fairness is guaranteed at the level of threads. A *thread* is a nonempty stack of tasks, where only the topmost task of a thread can be reduced. If the topmost task of a thread is reduced, it is replaced with the newly created tasks, if there are any. If a composition $E_1 \wedge E_2$ is reduced, the left expression E_1 goes on top of the right expression E_2, which means that E_1 is considered before E_2. If the topmost task of a thread is irreducible over the current store and the thread contains further tasks, the topmost task is moved to a newly created thread. A thread disappears as soon as it has no task left.



The outlined reduction strategy tries to be as sequential as possible and as concurrent as necessary. It will execute the task `{Fib 5 M}` sequentially in a single thread, thus requiring only linear space.

The concurrent execution of an expression E in a separate thread can be forced by writing

```
local Fire in if Fire=1 then E else true fi Fire=1 end
```

With threads it is straightforward to send messages sequentially through a port. If ports are defined as in Section 9, the simple composition

```
{Port A}  {Port B}
```

will send `A` before `B` provided the store binds `Port` already to a port.

17 Time

It is straightforward to extend OPM such that tasks can be synchronized on time points. Here we specify a timer primitive

```
{Sleep T X Y}
```

which equates two variables `X` and `Y` after the time span specified by the variable `T` has passed. The timer primitive first waits until the store binds `T` to an integer n. It then stays irreducible for further n milliseconds, after which it can be reduced to the constraint `X=Y`.

18 Encapsulated Search

Since OPM has constraints and logic variables, it will subsume the problem solving capabilities of constraint logic programming when extended with a nondeterministic choice combinator. However, a completely new idea is needed for encapsulating the resulting problem solvers into concurrent agents.

A nondeterministic choice combinator can be provided as an expression

$$E_1 \textbf{ or } E_2$$

called a *choice*. Choice tasks can only be reduced if no other task is reducible. If this is the case, a choice can be reduced by *distributing* the computation space into two spaces obtained by replacing the choice with its left and right alternative, respectively. The resulting search tree of computation spaces can be explored with a suitable strategy. If a tell operation announces failure, computation in the corresponding computation space is aborted. The leaves of the search tree are either failed or unfailed computation spaces. Unfailed leaves will contain the solutions of the problem being solved as bindings to certain variables.

While the outlined semantics for nondeterministic choice provides the expressivity of constraint logic programming, distributing the top level computation space is not compatible with the idea of concurrent computation. What we would like to have are concurrent agents to which we can present a search strategy and a problem to be solved and from which we can request the solutions of the problem one by one. This means that the search agent should encapsulate search. It turns out that such a search agent can be programmed with a single further primitive, called a *search combinator*. The search combinator spawns a subordinate computation space and reduces in case the subordinate space fails, becomes irreducible, or is distributed. In the case of distribution, the two alternative local spaces are frozen and returned as first-class citizens represented as procedures. The details of this elaborate construction are reported in [20, 19, 22].

The resulting model is realized in Oz together with further concurrent constraint combinators [20, 19, 14]. Oz gets constraint logic programming out of its problem solving ghetto and integrates it into a concurrent and lexically scoped language with first-class procedures and state. This integration eliminates the need for Prolog's ad hoc constructs and also increases the expressivity of the problem solving constructs.

19 Summary

We have presented a simple and expressive model OPM for high-level concurrent programming. The model is lexically scoped and consists of the concurrent constraint kernel OCC, first-class procedures, and cells providing for concurrent state. It computes with logic variables and constraints and monotonically synchronizes on a declarative constraint store.

The constraint store is the exclusive place where information about the values of variables is stored. Dynamically created values called names interface the constraint store with the procedure and the cell store. This way OPM realizes an orthogonal combination of first-order constraints with first-class procedures and stateful cells. We have shown how OPM can express higher-order functions, agents and objects. We have added an incremental tell operation to improve the potential for parallelism. We have also added propagators, threads, and a timer primitive as needed for a practical language. Finally, we have outlined how the model can be extended so that it can express encapsulated problem solvers generalizing the problem solving capabilities of constraint logic programming. Oz translates the presented ideas into an exciting new programming language.

Acknowledgements

The development of OPM and Oz would have been impossible without the combined contributions of the members of the Programming Systems Lab at DFKI. Much inspiration and technical knowledge came from the developers of AKL at SICS, the developers of LIFE at Digital PRL, and the other partners of the Esprit basic research action ACCLAIM.

I'm grateful to Seif Haridi, Martin Henz, Michael Mehl, Joachim Niehren, Andreas Podelski, and Christian Schulte who read and commented on drafts of this paper.

The research reported in this paper has been supported by the BMBF (contract ITW 9105), the Esprit Basic Research Project ACCLAIM (contract EP 7195), and the Esprit Working Group CCL (contract EP 6028).

Remark

The DFKI Oz system and papers of authors from the Programming Systems Lab at DFKI are available through the Web at `http://ps-www.dfki.uni-sb.de/` or through anonymous ftp from `ps-ftp.dfki.uni-sb.de`.

References

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

[2] R. Backofen. A complete axiomatization of a theory with feature and arity constraints. *Journal of Logic Programming*, 1995. To appear.

[3] L. Cardelli. Obliq: A Language with Distributed Scope. In *Proc. 22nd Ann. ACM Symposium on Principles of Programming Languages (POPL'95)*, pages 286–297, 1995.

[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] 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*, pages 27–48. The MIT Press, Cambridge, MA, 1995.

[6] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19/20:503–582, 1994.

[7] S. Janson and S. Haridi. Programming paradigms of the Andorra kernel language. In V. Saraswat and K. Ueda, editors, *Logic Programming, Proc. 1991 Int. Symposium*, pages 167–186. The MIT Press, Cambridge, MA, 1991.

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

[9] J. Launchbury. A natural semantics for lazy evaluation. In *Proc. 20th Ann. ACM Symposium on Principles of Programming Languages (POPL'93)*, pages 144–154, 1993.

[10] M. J. Maher. Logic semantics for a class of committed-choice programs. In J.-L. Lassez, editor, *Logic Programming, Proc. 4th Int. Conference*, pages 858–876. The MIT Press, Cambridge, MA, 1987.

[11] M. Mehl, R. Scheidhauer, and C. Schulte. An abstract machine for Oz. In *Proc. 7th Int. Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'95)*. Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1995. To appear.

[12] R. Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, Berlin, 1980.

[13] R. Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[14] T. Müller, K. Popow, C. Schulte, and J. Würtz. Constraint programming in Oz. DFKI Oz documentation series, DFKI, Saarbrücken, Germany, 1994.

[15] J. Niehren. *Funktionale Berechnung in einem uniform nebenläufigen Kalkül mit logischen Variablen*. Doctoral Dissertation. Universität des Saarlandes, Saarbrücken, Germany, December 1994. Submitted.

[16] J. Niehren and G. Smolka. A confluent relational calculus for higher-order programming with constraints. In J.-P. Jouannaud, editor, *Proc. 1st Int. Conference on Constraints in Computational Logics (CCL'94)*, pages 89–104. Lecture Notes in Computer Science, Vol. 845, Springer-Verlag, Berlin, 1994.

[17] G. D. Plotkin. A structural approach to operational semantics. DAIMI FN-19, Dept. of Computer Science, Aarhus University, Denmark, 1981. Reprinted 1991.

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

[19] C. Schulte and G. Smolka. Encapsulated search in higher-order concurrent constraint programming. In M. Bruynooghe, editor, *Logic Programming, Proc. 1994 Int. Symposium*, pages 505–520. The MIT Press, Cambridge, MA, 1994.

[20] C. Schulte, G. Smolka, and J. Würtz. Encapsulated search and constraint programming in Oz. In A. Borning, editor, *Proc. 2nd Int. Workshop on Principles and Practice of Constraint Programming (PPCP'94)*, pages 134–150. Lecture Notes in Computer Science, Vol. 874, Springer-Verlag, Berlin, 1994.

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

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

[23] G. Smolka. A foundation for higher-order concurrent constraint programming. In J.-P. Jouannaud, editor, *Proc. 1st Int. Conference on Constraints in Computational Logics (CCL'94)*, pages 50–72. Lecture Notes in Computer Science, Vol. 845, Springer-Verlag, Berlin, 1994.

[24] G. Smolka. Fresh: A higher-order language with unification and multiple results. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Relations, Functions, and Equations*, pages 469–524. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[25] G. Smolka. An Oz primer. DFKI Oz documentation series, DFKI, Saarbrücken, Germany, 1995.

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

[27] G. Winskel. *The Formal Semantics of Programming Languages.* Foundations of Computing. The MIT Press, Cambridge, MA, 1993.