Higher Order Programming for Unordered Minds

Juris Reinfelds

Klipsch School of Electrical & Computer Engineering,
New Mexico State University
juris@nmsu.edu

Abstract. In this paper we describe our experience with how Mozart-Oz facilitates the introduction of distributed computing to students of limited programming background and how the application of a few basic programming concepts can increase the students' comprehension of how distributed computations actually happen.

1 Introduction

In a graduate CS course in Spring 2003, there was a need to introduce distributed computing as quickly and concisely as possible. As it often happens in CS programs with students from all parts of the world, their background knowledge varied widely in scope and depth with hands-on lab-skills at a very low level.

Java with remote threads [1] or MPI [2] with C-programming was beyond the reach of most students. Instead we took advantage of the clear, concise and powerful semantics of Oz and Oz's natural inclusion of distributed computations. To illustrate how we fared, this paper will discuss the simple but canonical Compute Server/Client Problem of distributed computing.

First, in Spring 2003, we took the conventional approach as illustrated in the Mozart Documentation [3] and in Van Roy & Haridi [4]. The students quickly learned how to save and take tickets and where to put "their code" in the preparation of client's compute tasks. This enabled the students to complete the required homeworks, but their depth of understanding and interest in the power and possibilities of this new kind of programming remained low. Only one of 21 students of this course saw the power of the Mozart approach, learned more Oz and applied it to other projects in other courses. Section 2 describes this approach and defines our version of the Compute Server/Client Problem.

To improve the students' depth of comprehension, we set out to determine what programming concepts and mechanisms underlie the remarkable simplicity, directness and ease with which distributed computations can be initiated and controlled in Oz. In our opinion, the key to simple yet powerful distributed computing lies in the distributed, internet-wide value store of Oz. We developed a programmer's model of such value storage and management. This is discussed in Section 3.

Using our new approach, half the class of 18 students in Spring 2004 was sufficiently interested and able to use Mozart in their end of semester project although they had the freedom to choose any programming tools and methods and they had a wide range of project topics available to them. In Section 4 we augment the Compute Server/Client we introduce in Section 2 with an explanation of how it works in terms of the concepts that we define in Section 3.

In conclusion, we suggest that distributed computations would be facilitated even more if Mozart could provide an option to access a global value store as a network service by extending the existing mechanisms with which the runtime system of Oz accesses remote values.

2 A Simple Compute Server and Client

Let us define a simple compute-server and client by removing the unessential object orientation from the compute-server example that appears in Mozart documentation [3] volume "Distributed Computing", Section 3.2.4 as well as in VanRoy & Haridi [4] Chapter 11.

To further simplify the surprisingly concept-rich program, we have omitted exception message handling in the code that we give to the students. We ask them to insert exception handling into the given code in a homework assignment to expand and test the depth of their understanding of Oz programming.

2.1 The Server

The server uses an Oz-Port *(Port)* to collect incoming messages from one or more clients into a list *(PortList)*. The Port mechanism appends incoming messages at the end of the port list and maintains an unbound identifier at the end of this list. Processing of the list suspends when it reaches the unbound identifier until that identifier gets bound to a value which is the next incoming task.

The server expects incoming messages to refer to Oz-values that contain zero-arg-procedures and sets up a recursive *ForAll* loop that executes each zero-arg-proc from the head of the list until execution suspends on the unbound identifier at the end of the list. Here is the Oz program for our version of the server:

```
% Server waits for and processes compute tasks
% that are zero-arg-procedures.
proc {ComputeServer}
   PortList     % list of incoming tasks
   Port                 % appends incoming tasks to PortList
   Q            % one-arg-proc for recursive loop
   TicketToPort         % offers remote access to Port
in
   Port = {NewPort PortList}
   TicketToPort = {Connection.offerUnlimited Port}
   {Pickle.save TicketToPort "/home/juris/filenameOfTicket"}
   proc {Q I} {I} end  % if incoming I's are zero-arg-procs
   {ForAll PortList Q}
end %ComputeServer
```

Here it is again. Expressed more concisely and parametrized for multiple server creation the server program looks simpler than it is:

```
proc {CompSrv FileName}
   PortList
   Port={NewPort PortList}
in
   {Pickle.save {Connection.offerUnlimited Port} FileName}
   {ForAll PortList proc {$ I} {I} end}
end %CompSrv
```

Students had no trouble with typing-in and executing this program to create one or more compute servers, yet understanding how the server works is another matter that we will take up in Section 4. At this stage students cannot understand open lists and external variables and have difficulties with the simplest modifications of the server program. For example, a simple modification to enable examination of the PortList while the server runs is beyond the grasp of most students because the idea that PortList could be an external variable in CompSrv needs to be introduced gently to students brought up on the string-of-pearls model [5] of side-effect avoidance in imperative programming.

2.2 The Ackermann Function

The Ackermann function is an example of a very simple code that defines non-trivial computations of any desired duration. In the days of early mainframes and compilers the Ackermann Number was used to measure the recursive capabilities of mainframe-op.sys.-compiler combinations [6]. The Ackermann Number is defined as the smallest value N for which Ack(3,N) crashes because some resource in the computer-operating system-compiler combination runs out. In the testing of our compute server we can choose suitable values of N to give a run time of a few minutes, so we can use an Oz-panel on each machine to observe the remote computations as they happen. Here is the code of the Ackermann function. The arguments are small non-negative integers.

```
fun {Ack M N}
   if M==0 then N+1
   elseif N==0 then {Ack (M-1) 1}
   else {Ack (M-1) {Ack M (N-1)} }
   end
end %Ack
```

2.3 The Client

The conventional explanation of the compute-client goes somewhat like this. Any Oz-invocation can become a client of our server if it has access to the file that stores the Oz-ticket to the server's Oz-port. To become a client an Oz-invocation has to *Pickle.load* the "pickled" ticket into the client's Oz invocation and then *Connection.take* the server's port-ticket to establish a connection to the server's port. The client can create a compute-task by taking a statement sequence in client's name-space and wrapping it into a zero argument procedure. Then the client uses the built-in *Send* procedure to send the zero-arg-proc to the server's port for processing. The Oz-runtime systems of client and server manage the network connections and transmissions between client and server in a program-transparent way.

Here is a statement by statement discussion of the client's Oz-code. The potential client becomes a client by establishing a connection to the server's Oz-port by executing

```
ServPort1 = {Connection.take {Pickle.load "filenameOfTicket"}}
```

Without a model of the global store students find it difficult to reason about the ticket mechanism. For example, is the connection made by transferring the remote value or its global store reference? Suppose that the client wants to calculate the value of Ack(3,18) remotely. In other words, the client wants to execute the following statement on the server:

```
M3N18 = {Ack 3 18}
```

The client constructs a zero argument procedure:

```
proc {ZeroArgProc}
   M3N18 = {Ack 3 18}
end %ZeroArgProc
```

sends it to the server for processing and displays the result in the Browser window at client:

```
{Send ServPort1 ZeroArgProc}
{Browse m3n18#M3N18}
```

and it works! The user is pleased but confused. Just how did all this happen? The client did not explicitly tell the server what *Ack* was. What if the server already has the identifier Ack bound to a different value? The calculation of Ack(3,18) makes millions of calls to the function Ack that is defined on the client but not on the server. Did our remote computation swamp the network?

A deeper understanding of the programming concepts and mechanisms that underlie our compute server is necessary to answer these questions. In the rest of the paper, we will explore one way to achieve more depth in minimal time even if students have a limited background in programming and mathematics.

3 Programming Concepts and Mechanisms on Which Compute Server Is Based

First we define several basic programming concepts especially where we differ from historically established traditional definitions. Then we define a programmer's model of a distributed, global value store which in our opinion is the key to a deeper understanding of why Oz achieves distributed computations so simply and naturally.

3.1 Definitions

Statement. An *Oz-statement* is a piece of information that defines one step in the transformation of input information toward a desired output. An Oz-program is a sequence of Oz statements.

Consequences of this definition: since a program is a sequence of statements, execution should start with the first statement of the program and proceed with the next statement until there are no more statements to execute. Since a statement defines one step, we should be able to compile and execute one statement at a time. Imposition of "main program" or function or method named "main()" is an unnecessary restriction on what the user may want to do.

Value. An *Oz-value* is *any piece of information* of a type that the Oz programming language can handle. The *type* of a value defines a set of properties that are common to all values of that type. In particular, a type defines which operators of a programming language can accept values of this type as arguments. Acceptable Oz-value types are int, char, procedure, function, class, object and others. The programmer is burdened with the least amount of semantic baggage if all values can be handled in the same way as much as possible.

Consequences of this definition: The historically established requirement for special handling of procedure and function-value introductions is an unnecessary restriction on the programmer. An introduction of integer 256 creates an Oz-value of type "int" and an Oz-value of type "function" is introduced by executing the Oz-declaration

```
fun {$ N} N*N*N end
```

which introduces a function that returns the cube of its argument. It should and does behave very much like an Oz-value that is an integer. We can form an expression with it where it is called with the argument 3 and the expression returns the value 283.

```
256 + {fun {$ N} N*N*N end  3}
```

Bowing to years of traditional practice, Oz also accepts proc/fun value creation combined with identifier binding as in

```
fun {Cube X} X*X*X end
```

Regardless of which form of introduction we choose, function introduction creates a value that is just another piece of information. Only when a function is executed, the function value is used to create an information transformation process that produces the desired result. With such an introduction of procedure and function values, higher order programming becomes the default without the need to explain to mathematically naive students what "higher order" is.

3.2 One Distributed Value Store for All Invocations of Oz

Here we will describe a programmer's model of a global value store that combines the design ideas of the declarative Oz-value store and the the design concepts of the Unix file system. We will use programming concepts that best allow a programmer to relate this model to program design and structure. Our aim is to provide a model that truthfully portrays the behavior and capabilities of the Mozart value store but that does not necessarily reflect every nuance of its implementation.

A key concept is that there is one universal, distributed value store. The runtime system of every Oz invocation maintains its piece of the universal value store of Oz. In other words, all invocations of Oz, past, present and future, maintain one distributed data base of Oz-values.

Each invocation of Oz maintains the Oz-values that were created in this invocation and which are still useful. An Oz *value is useful* if there is at least one valid Oz identifier in any Oz-invocation that references this value.

Traditional programming languages adhere to the implementation-inspired definition:

> *a variable is both a name that corresponds to the address of a memory location and a value at that memory location.*

In traditional programming languages we usually have a one to one correspondence between variable names and the values to which the names are bound (or as we say, assigned). To bind another name to a value that already has a name assigned to it, we use pointers or a similar mechanism. We teach ourselves to say that after the assignment `X:=5` "X *is* five" and in doing so we attach a second meaning to X which already meant *a memory location containing int.* In the statement `X:=X+1`, we quietly overlook the de-referencing of one X, but not the other, hoping that the compiler will get it right.

Oz makes a clear separation between the local name space and scope of identifiers that programmers use in Oz-programs and global value store references that must uniquely distinguish between values from all Oz-invocations.

The binding statement of Oz supports variable-value binding as in traditional languages. It also supports variable-variable bindings so that both variables become bound to the same value. This removes the need for a pointer mechanism, but requires the introduction of a value-type "reference" and programmer-transparent de-referencing by the runtime system to explain the binding of additional identifiers to a value that is already bound to an identifier.

Oz supports the introduction and use of unbound variables. Our model accommodates unbound variables by introducing an additional value-type which we call *"no-value-yet"*. A store-item with type "no-value-yet" is created for every identifier at the moment this identifier is introduced. Similarly, whenever a value is introduced a store-item is created containing the value and its type. Store items are identified by globally unique reference tags. Binding statements link store-items of identifiers to store-items of values.

To remind us of the differences between Oz and traditional programming languages, we use the term "binding" instead of "assignment" and the terms "identifier" and "value" instead of "variable" which in conventional languages often denotes either the storage address allocated to that variable or the value that is stored there. In our model a global value store item has three components:

– a reference tag that is unique in all invocations of Oz;
– the type of this Oz-value;
– the bit-pattern that represents this Oz-value.

When a new identifier is introduced in the program, the compiler creates a global value store item with a new unique reference and with the type "no-value-yet". The compiler also updates its name table with an entry that relates the newly introduced identifier with its value store reference. The compiled byte-code of this and subsequent statements within the scope of this identifier will contain the globally unique reference to the value store item instead of the local identifier. In this way local name-space remains strictly local and unconcerned if same identifiers reference different values in other name spaces, while byte-code can be executed to give the same result in any context in any invocation of Oz.

Identifier introductions are executable statements in Oz, but the scopes of identifiers are known at compile-time from actual and implied `local ... in ... end` statements. Formal parameters of proc/funs are treated as belonging to a `local ... end` statement around the proc/fun body that includes the call of this proc/fun in the scope. This scope is somewhat unusual, but it turns the formal/actual parameter mechanism of a proc/fun call into a sequence of binding statements of formal parameters with their corresponding actual parameter expressions.

Internal `local ... in ... end` statements introduce identifiers that are scoped within the proc/fun's body. Other identifiers that appear in the procedure or function are called *"external variables"*. They must already have global store references and the compiler places their global store references into the Oz byte-code, so that the code can be executed in any context in any Oz-invocation with the same results.

Oz has no restrictions as to where procedures and functions may be introduced in a program or expression. In this sense the handling of procedure and function values is no different from the handling of integer and char values which are also compiled into their value store items when and where they appear in the program.

Store references may appear in store items as values with type "reference". Appropriate dereferencing is performed by the executing runtime system. One situation where a reference must appear as the value of a value store item is for an identifier that is bound to an Oz-cell. In the statement that creates a cell (a state variable that can change the value to which it refers) we have

```
local C in {NewCell 25 C} ... end
```

First, the identifier C is introduced with the type "no-value-yet". The execution of the procedure NewCell *binds C to a cell*, which in terms of our model means that NewCell

– Creates a new store item containing int 25.
– Changes the type of store item of C from "no-value-yet" to "cell".
– Sets the value of C's store item to the reference to the newly created store item containing int 25.

In summary, the compiler compiles every identifier to a store reference, so that although we say that *"we send a compute task to Oz-port P"* when we execute the procedure call `{Send P Task}`, what actually gets transferred to the input queue of P is a reference to the store item represented by the identifier *Task*.

3.3 Global Value Store as a Network Service

The current implementation of the global value store works very well if all participating Oz-invocations stay up as long as needed. However, the handling of situations when a remote site has shut down making its global value store references unexpectedly invalid is extremely awkward. For a few references this can be remedied by pickling a ticket for each reference, so that on restart a new reference can be obtained more easily. For more connected distributed computations as well as for backup and data preservation it might be useful to provide at least a part of the global value store as a *network service* using currently well developed database and uninterruptable hardware technology that would provide backup and redundancy 24/7 with continuous availability and recovery of previous states of computation if so desired. On the surface, it seems that the cost of the extra network accesses would be well worth it, especially because the current remote value fetching mechanism is so effective and efficient. There is hope that relatively small extensions of the current mechanisms to provide such a network service might provide a substantial facilitation of distributed computations.

There is active Mozart research along these lines [7], but it is focused on management of fault-tolerance using ordinary machines as components. We believe that an uninterruptable global value store would be a valuable network service for Mozart users, especially when applying Mozart to cluster computing on medium to large clusters.

3.4 External Variables in Procedures and Functions

Here we summarize the concept of external variables of Oz. At the point of introduction of a procedure or function there may be previously introduced identifiers that are in scope. If these identifiers appear in the procedure that is being introduced, the compiler will compile their store references into the Oz-byte-code of the procedure. These identifiers and the values to which they refer are called "external variables" of the procedure. Whenever a procedure value is bound to an identifier, the lifetime (scope) of the procedure value and the lifetime of the value store items to which its external variables refer is extended by the lifetime of this identifier.

Our model creates the same semantics for declarative external variables as the more mathematical environments of VanRoy&Haridi [4] Section 2.4, but our model of the global value store includes Oz-cells and brings the external variable concept closer to the experience of programmers who have followed the traditional Algol-Pascal-C pearls-of-a-necklace of Dijkstra [5] style of program design and who have regarded any departure from strictly nested stack-implementable variable scopes such as the relatively modest *"own variables"* of Algol-60 [8] as unacceptable.

4 The Compute Server/Client in Terms of Our Value Store Model

Identifiers are introduced by explicit or implicit

```
local Identifiers ... in StatementSequence ... end
```

local-statements that determine the scope of each identifier. In our model the compiler associates each identifier with a unique global reference tag as it processes the left part of the local-statement. The compiler compiles global references and not local identifier names into byte code. At execution time the runtime system executes the byte code and creates a store item with the reference of this identifier and the type *no-value-yet*.

Values are introduced as text in the source code and byte code is compiled requesting the runtime system to create a store-item with a unique reference tag and appropriate type and value.

Binding statements bind store-items of identifiers to store-items of values by placing the reference of the value into the value part of the identifier's store item with type "reference". When execution requires actual values, but the store item of the identifier contains a value of type reference, the runtime-system dereferences values of type "reference" until a non-reference value is reached and performs dynamic type-checking to ensure that this value is compatible with the operator that requires it.

4.1 The Server

Let us discuss the global store and identifier scope aspects of the compute server.

```
proc {CompSrv FileName}
```

This is an abbreviation for a binding statement within some local-statement such as

```
local
   CompSrv
in ...
   CompSrv = proc {$ FileName}...end
   ...
end
```

where the formal parameter *FileName* has an implied `local FileName in ...` that includes the proc-call *and* the procedure body, so the identifier of this formal parameter, which appears in the procedure body, can be bound to its corresponding actual parameter value when the proc is executed. This removes the need for a special consideration of formal-actual parameter mechanisms. Continuing with the statements of the server:

```
proc {CompSrv FileName}
PortList Port in
```

There is an implied *local* before PortList that scopes the identifiers PortList and Port to the procedure body. Next:

```
Port = {NewPort PortList}
```

The identifier NewPort is external to this procedure. It does not appear as a formal parameter nor is it defined as a procedure-local identifier. If there is no lexically more recent local use of the identifier NewPort, the compiler compiles a call to the system function NewPort.

The identifier NewPort and its store reference were placed into the compiler's identifier table as this Oz-invocation started up. This applies to all system provided procedures and functions. Next,

```
{Pickle.save {Connection.offerUnlimited Port} "filename"}
```

A text-string, called ticket, which allows remote invocations of Oz to pick up the global store reference to the server's Oz-port, will be placed in a file in the current directory from which the server proc is executed. Next,

```
{ForAll PortList proc {$ I} {I} end}
```

The second argument of this call of the system-supplied procedure *ForAll* is a procedure value **proc ... end** that is introduced where it appears in the procedure-call. This procedure value (actual parameter) gets bound to the corresponding formal parameter of *ForAll*, so it can be used in the procedure body. It cannot be used anywhere else in the program because it is not bound to an identifier that is valid outside of the procedure body.

We also need to explain the Oz-Port mechanism and how ForAll works. Let us use source code statements to describe how the execution proceeds:

```
Port = {NewPort PortList}
```

There is very little information about how Oz-ports and Oz-runtime systems work. According to VanRoy&Haridi [4] p.719, an Oz-Port is a value that is a FIFO channel with an asynchronous Send-operation that allows many-to-one communications from multiple clients. NewPort needs an unbound identifier (we use PortList) as argument which NewPort binds to the list of incoming global value store references that clients will *Send* to this port. The type"port" value returned by the NewPort call gets bound to our unbound identifier *Port*. The port mechanism terminates the list *PortList* with an unbound identifier. The port mechanism ensures that each newly arrived Send-carried input extends the list with itself and a new unbound identifier. In this way, if we consume the list in the usual way from the head, we have a FIFO queue of incoming items in PortList.

```
{Pickle.save {Connection.offerUnlimited Port} ´filename´}
```

System function Connection.offerUnlimited returns an ASCII string version of the global store reference of the argument of the call. System procedure

Pickle.save writes the ASCII string value of its first argument into a file provided that the second argument is a valid file name on the platform on which this Oz-invocation runs. This is a convenient way to convey the global value store reference of the Oz-Port of the Compute Server to a number of potential clients in systems where NFS is used to share the same home directory over a cluster of computers.

```
{ForAll PortList proc {$ I} {I} end}
```

`ForAll` executes the second argument that should be a one-argument procedure value with each element of PortList as the argument. Since Oz expressions suspend execution if execution reaches a value of type “no-value-yet”, `ForAll` waits at the end of the input list for the arrival of input items which it executes, in order of arrival, and then waits again. From the form of the procedure value in the ForAll call, we see that our server will execute incoming zero-argument procedure values. Any other incoming value will raise an exception which in this very simple version of the server will not get transmitted back to the client.

4.2 A Client

Assume that a potential client has introduced four identifiers

```
Ack M3N18 ServPort1 CompTask
```

and Ack is bound to the Ackermann function that we discussed in a previous section. To become a client of our compute server, we have to pick up the global value store reference to the server’s Oz-Port that the server pickled into the file with the name “filename.” Assuming this filename is valid where this invocation of Oz is executed, the client executes the binding statement

```
ServPort1 = {Connection.take {Pickle.load ´filename´}}
```

Pickle.load returns the ASCII string version of the value store reference of the Oz-Port of the server. *Connection.take* converts the ASCII string version to an actual value store reference that gets bound to ServPort1, so that after this statement completes the value store item of ServPort1 on the client contains the value store reference to the Oz-Port of the server with the value-type “reference.” Whenever the client program needs the actual port-value, as in a Send call, the runtime system of the client will recognize that the reference is remote and in collaboration with the runtime system of the server will obtain access to the actual port-value. The Oz-Port value at the server is now bound to two identifiers Port and ServPort1, which are in separate namespaces on separate invocations of Oz.

Client creates a zero-arg procedure that defines a computation task. The identifiers M3N18 and Ack are external variables in the zero-arg procedure. The identifiers Ack and M3N18 are local to the client, but their global value store references ref(M3N18) and ref(Ack) are compiled into the zero-arg-procedure value that is sent to the server.

```
CompTask = proc {$} M3N18 = {Ack 3 18} end
{Send ServPort1 CompTask}
```

The system procedure *Send* arranges help from the runtime systems of the client and the server to use the network connection between these runtime systems that was established when the ticket to the server's Port was taken. "Send" transfers the value store reference of CompTask from client to server-port Port. The server-port appends this reference to the end of its input list. Although we like to say: *"We send a compute task to the server"*, only the global value store reference ref(CompTask) gets attached to the end of the server's ProcList. If and when the execution of some expression on the server (as in the ForAll call) requires the value itself, the server's runtime system will use the remote reference to get to its Oz-value. The Send procedure terminates when the network transfer of the reference is completed.

In our server, the incoming reference creates a value store element of type "reference" that becomes a list element in the input list and does not have its own identifier in the identifier table of the server.

The ForAll procedure fetches the remote value from the client and executes it as a zero argument procedure. If the remote value is not a zero arg proc, an exception is raised and our very simple server crashes. Programming of crash avoidance is a good exercise for the students.

During the execution of the remote ref(CompTask), the runtime system of the server encounters two more remote references for M3N18 and Ack. If the runtime system of Oz is implemented efficiently, the server copies the value of Ack from the client calls it locally millions of times. The server determines that the value of M3N18 is of type "no-value-yet" and binds it to a type "int" value that is the result of the computation. The client can now see the result of the remote computation because it is bound to the client's local identifier M3N18.

5 Conclusions

Our model of the global store helps Oz programmers to untangle and visualize the way in which computations take place when values are referenced from several different name spaces by identifiers with non-nested, non-overlapping scopes. The negative history of Algol-60's modest break with stack-friendly scopes with "own variables" [8], [9] shows that scopes that are not cleanly nested and therefore stack-implementable are a hard nut for imperative programmers and compiler writers. It is wonderful that Oz has cracked this nut so cleanly, elegantly, and effectively.

It is interesting to observe that the design of our model is similar to the design of the UNIX file system with

– value <> i-node,
– identifier <>file-name or link-name
– identifier-table <>directory.

Cluster computing and well connected distributed computations in general would benefit greatly if the existing remote value handling mechanisms of Oz could be extended to provide the global value store as a 24/7 network service with backups, crash-proof redundancy and optional restoration of previous states of computation.

Acknowledgements

The author is grateful to the students of CS5340 at UTEP in Spring 2003 and Spring 2004 for their patience, understanding and willingness to try new ideas while our global value store model was under development. The author is grateful to Denys Duchier for the rescue of our Mozart system in early 2003 when the Emacs-based API refused to run because SUSE Linux 8.1 had not loaded a font that no one needed, but nevertheless Emacs decided to check for its existence before it allowed the Oz-menu to be created although Oz did not need this font either. The author is grateful to the authors of the Mozart-Oz system, especially to Peter Van Roy for many discussions on the deeper aspects of the Kernel Language and Oz design.

References

1. Oaks, Scott,& Wong, Henry, *Java Threads*, Second Ed. O'Reilly (1999).
2. Pacheco, Peter S., *Parallel Programming with MPI*, Morgan Kaufmann Publishers Inc. (1997).
3. Mozart-Oz: web site and home-page: http://www.mozart-oz.org (2004).
4. Van Roy, Peter., Haridi, Seif, *Concepts, Techniques, and Models of Computer Programming*, MIT Press (2004).
5. Dijkstra, Edsger W., *A Necklace of Pearls*, p.59, Section 14, Structured Programming, Academic Press (1972).
6. Sundblad, Yngve, *The Ackermann Function, a Theoretical, Computational and Formula Manipulative Study*, BIT, Vol. 11, p. 107-119 (1971).
7. Al-Metwally, Mostafa, Alouini, Ilies, *Fault Tolerant Global Store Module*, http://www.mozart-oz.org/mogul/doc/metwally/globalstore (2001).
8. Naur, Peter, Editor: *Report on the Algorithmic Language Algol 60*, CACM, Vol. 3, #5, p. 299-314, (1960).
9. Wirth, Niklaus, *Computing Science Education: The Road Not Taken*, SIGCSE Bulletin 34(3), 1-3., (2002)