Logic Programming in Oz with Mozart

Peter Van Roy
Université catholique de Louvain
B-1348 Louvain-la-Neuve, Belgium
pvr@info.ucl.ac.be

This short tutorial explains how to do Prolog-style logic programming in Oz. We give programming examples that can be run interactively on the Mozart system, which implements Oz. The Oz language is the result of a decade of research into programming based on logic. The Oz computation model subsumes both search-based logic programming and committed-choice (concurrent) logic programming with deep guards. Furthermore, Oz provides new abilities, such as first-class top levels and constant-time merge, that exist in neither of its ancestors. We show two of the Oz interactive graphic tools, namely the Browser and the Explorer, which are useful for developing and running logic programs. We conclude by explaining why logic programming is just a prelude to the real strengths of Oz, namely constraint programming and distributed programming. In these two areas, Oz is equal to or better than any other system existing in the world today. For example, for compute-intensive constraint problems Oz provides parallel search engines that can be used transparently, i.e., without changing the problem specification.

1 Introduction

The Oz language makes it easy to write efficient, declarative logic programs that combine the advantages of search-based logic and constraint languages (such as Prolog, CHIP, and cc(FD) [16, 7]) and committed-choice (concurrent) logic languages (including flat languages such as Parlog, FCP, and FGHC, but also deep languages such as Concurrent Prolog and GHC [14]). Furthermore, Oz provides powerful new programming techniques that are not possible in either language family.

This tutorial explains the basic ideas of Oz by means of small examples that run on Mozart, a recently-released system that efficiently implements the latest version of Oz, also known as Oz 3 [8]. Mozart has an easy-to-use interactive user interface based on Emacs. We suggest that you download the Mozart system and try the examples at the keyboard. Only a few basic keyboard commands are needed to use

Mozart interactively; see the Oz tutorial for more information [4].

The purpose of this tutorial is to be an entry point into the Oz universe for people with some Prolog experience. We assume that you understand the basic concepts of Prolog and that you have written or understood some small Prolog programs. We show that the step from Prolog to Oz is not hard at all, and we try to explain some of the vastly more general and powerful operations that are possible in Oz.

This tutorial focuses on logic programming for general-purpose applications that require manipulating structured data according to logical rules, i.e., applications such as rule-based expert systems or compilers. The Oz support for logic programming just scratches the surface of the real strengths of Oz, which are constraint programming and distributed programming. It is not the purpose of this tutorial to investigate these two areas, but we invite the interested reader to look at them. There exist excellent tutorials on constraint programming in Oz, e.g., with finite domains [13], with finite sets of integers [9], and for natural language processing [3]. There also exists a tutorial on distributed programming in Oz [18].

This tutorial is structured as follows. Section 2 shows how to do deterministic logic programming in Oz, i.e., sequential logic programming without search. This section also presents the Browser, a graphic tool for examining data structures. Then Section 3 extends this to nondeterministic logic programming, i.e., including search. This section also explains how to use first-class top levels, and it presents the Explorer, a graphic tool for interactive exploration of the search tree. Section 4 shows how to do committed-choice logic programming in Oz. Then Section 5 shows how search-based and committed-choice logic programming are combined. Section 6 summarizes the logic programming support in the Oz kernel language. Section 7 gives some glimpses into constraints and distribution in Oz. Finally, Section 8 concludes and gives perspectives on future developments.

2 Deterministic Logic Programming

Oz supports deterministic logic programming with three statements: **`if`**, **`case`**, and **`cond`**. The following example defines an `Append` predicate that can be used to append two lists:

```
declare
proc {Append L1 L2 L3}
   case L1
   of nil then L2=L3
   [] X|M1 then L3=X|{Append M1 L2}
   end
end
```

This example introduces two important syntactic short-cuts of Oz. In a **case** statement, variables in the branches of the case (like `X` and `M1`) are declared implicitly and their scope covers one branch of the case. Any procedure, e.g., `Append`, can be syntactically used as a function. The procedure's last argument is the function's output and is syntactically hidden.

This definition looks very much like a standard functional definition of append, but it is in fact much more general. The definition has a precise logical semantics in addition to its operational semantics. The logical semantics are:

$$
\begin{aligned}
&\forall l_1, l_2, l_3 : \mathrm{append}(l_1, l_2, l_3) \;\leftrightarrow \\
&\quad l_1 = \mathrm{nil} \wedge l_2 = l_3 \vee \\
&\quad \exists x, m_1, m_3 : l_1 = x|m_1 \wedge l_3 = x|m_3 \wedge \mathrm{append}(m_1, l_2, m_3)
\end{aligned}
$$

The operational semantics of the **case** statement are as follows. The **case** statement waits until its input is sufficiently instantiated to decide that one branch succeeds and all previous branches fail. If all branches fail then it takes the else branch or raises an exception if there is none. For `Append` this means that `L1` must be bound to a list. If `L1` has an unbound tail, then execution blocks until the tail is bound.

Variable declaration and variable scope are defined quite differently in Oz and Prolog. In Prolog, both declaration and scope are defined *implicitly* through the clausal syntax. Namely, variables in the head are universal over the whole clause and new variables in the body are existential over the whole clause body. In Oz, declaration and scope are defined *explicitly*. The scope is restricted to the statement in which the declaration occurs. This is important because Oz is fully compositional (all statements can be nested). Oz has syntactic support to make the explicit declarations less verbose.

2.1 The Browser

Here's one way to execute `Append` and display the result:

```
declare A in
{Append [1 2 3] [4 5 6] A}
{Browse A}
```

which can also be written as follows:

```
{Browse {Append [1 2 3] [4 5 6]}}
```

This uses the same syntactic short-cut as the definition of `Append`. Note that elements of Oz lists are not separated by commas as in Prolog. This code displays the output of `Append` in the *Browser*, a graphic tool for examining data structures and observing their evolution [10]. The Browser is fully concurrent and it can display any number of data

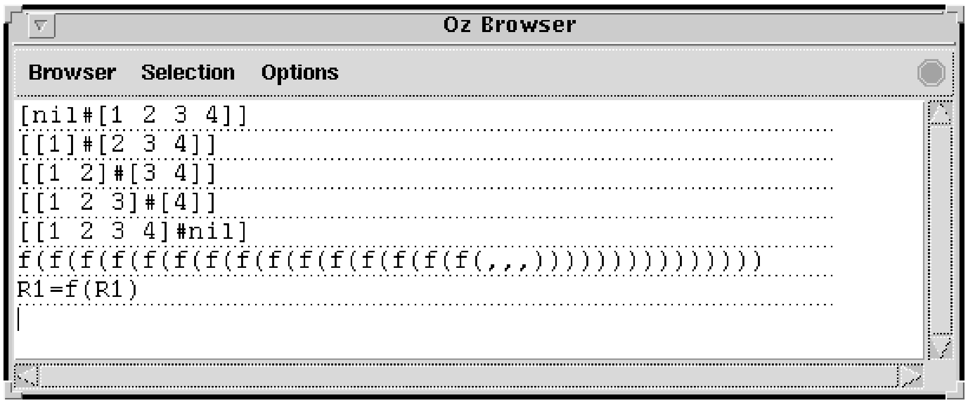

Figure 1: Screen shot of the Oz Browser.

structures simultaneously (see Figure 1). The display of a data structure containing unbound variables is updated when one of the variables is bound. The Browser has options to let it either ignore sharing or display sharing. In the second case, shared subterms (including cycles) are displayed only once.

Figure 1 gives a screen shot of a Browser window that shows the five solutions of a nondeterministic append (see next section) as well as two displays of:

```
declare X in {Browse X=f(X)}
```

The first displays `X` as a tree, stopping at the default depth limit of 15. The second displays `X` as a minimal graph, which makes all cycles and sharing explicit. You can set up the Browser to display the minimal graph by selecting Display Parameters in the Options menu.

3 Nondeterministic Logic Programming

Oz supports nondeterministic logic programming through two concepts: disjunctions (**dis** and **choice**) and first-class top levels. The **dis** and **choice** disjunctions do a don't-know choice, i.e., they can be used for search.

Let's write a nondeterministic version of append, i.e., one that can be used with many call patterns, similar to Prolog's:

```
declare
proc {FullAppend L1 L2 L3}
   dis L1=nil L2=L3
   [] X M1 M3 in
       L1=X|M1 L3=X|M3 {FullAppend M1 L2 M3}
```

```
      end
   end
```

The "`X M1 M3 in`" declares new variables `X`, `M1`, and `M3` for the second branch of the disjunction.

The **`dis`** statement is a determinacy-directed disjunction. If all clauses fail except for one, then execution continues with that clause. In this case execution is deterministic. If more than one clause is left, then a choice point is created and a top level is needed to continue execution. In this case execution is nondeterministic. This style of constraint propagation is also known as the *Andorra principle*. The **`choice`** statement is more primitive than **`dis`**; it generates a choice point immediately for all its clauses without checking if any clauses fail.

Both `Append` and `FullAppend` have exactly the same logical semantics. They differ only in their operational semantics. If used inside a top level, `FullAppend` gives results in cases where `Append` blocks.

3.1 First-Class Top Levels

The definition of `FullAppend` can be executed immediately, if used in a purely deterministic way:

```
{Browse {FullAppend [1 2] [3 4]}}     % Shows [1 2 3 4]
```

But the definition blocks if used nondeterministically:

```
{Browse {FullAppend X Y [1 2 3 4]}}  % Blocks
```

To get an answer, you need to execute the nondeterministic call in a top level. Here's how to create a top level with a given query:

```
declare
proc {Q A}
   X Y
in
   {FullAppend X Y [1 2 3 4]} A=X#Y
end
S={New Search.object script(Q)}
```

This defines a new procedure `Q` (the query) and a new object `S` (the top level) of class `Search.object`. The `Search.object` class is part of the `Search` module [2]. The class makes it possible to create any number of top levels. The top levels are accessed like objects, can run concurrently, and can be passed as arguments and stored in data structures. We say the top levels are *first class*. Each top level is initialized with a query. The query is entered as a one-argument procedure called a *script*. In this example, procedure `Q` contains the query. The procedure's output is `A`, i.e., the pair `X#Y`.

Creating a top level can be written more concisely as:

```
declare
S={New Search.object script(
   proc {$ A} X#Y=A in {FullAppend X Y [1 2 3 4]} end)}
```

This exploits two syntactic short-cuts. First, the "`$`" is a nesting marker that implicitly declares the variable `Q`. Second, putting an equation left of **`in`** implicitly declares all variables of the equation's left-hand side. That is, "`X#Y=A` **`in`**" declares `X` and `Y`, creates the pair `X#Y`, and unifies the pair with `A`.

You can get answers one by one by calling `S` as follows:

```
{Browse {S next($)}}
```

Each call `{S next($)}` gives a new answer. As before, the "`$`" is a nesting marker that implicitly declares a variable. The `next` method of object `S` is used to get the next answer. This is similar to the semicolon ";" in an interactive Prolog session. It is not identical, since the `next` must be called to get the first answer.

How do we know when there are no more answers? In a very simple way: each answer `A` is returned as a one-element list `[A]`. When there are no more answers, then `nil` is returned. So repeatedly asking for answers displays:

```
[nil#[1 2 3 4]]
[[1]#[2 3 4]]
[[1 2]#[3 4]]
[[1 2 3]#[4]]
[[1 2 3 4]#nil]
nil
```

There are five answers, from `nil#[1 2 3 4]` to `[1 2 3 4]#nil`. All further requests for answers give `nil`.

We conclude the discussion of top levels with a few random remarks:

- Creating a new top level is very cheap; you should not hesitate to do so for each query.

- A program can consist of deterministic and nondeterministic predicates used together in any way. A top level script can call such a program; this is possible because both deterministic and nondeterministic predicates have logical semantics. Of course, only the nondeterministic predicates can create choice points.

- It is easy to add information to an existing top level while it is active. It suffices for the script to have an external reference, i.e., to have a reference to something outside of the top level.

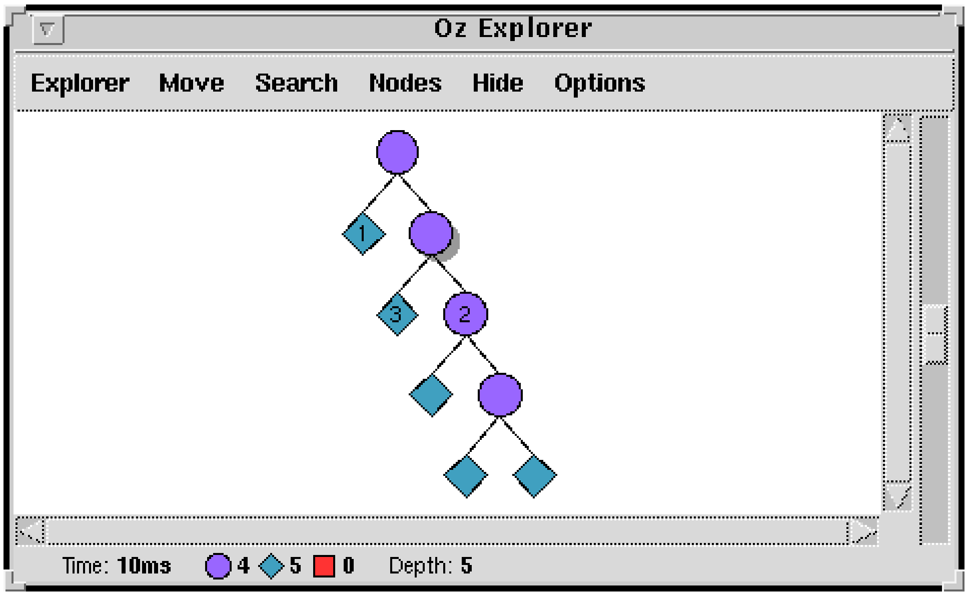

Figure 2: Screen shot of the Oz Explorer.

3.2 The Explorer

In addition to first-class top levels, another way to execute a logic program is by means of the *Explorer*, a graphic tool for interactive exploration of the search tree [12]. The Explorer was designed for constraint programming applications, but it is also very useful for logic programming. Here we show only a very small part of what the Explorer can do. To go further, we suggest that you try out the Mozart constraint demos with the Explorer.

The Explorer is an object that is given a script. Let's do this with the same `FullAppend` query as in the previous section:

```
{Explorer.object script(
 proc {$ A} X#Y=A in {FullAppend X Y [1 2 3 4]} end)}
```

This opens a window that displays the search tree. Initially, just the root is displayed, as a gray circle. The circle means that the root has a choice point. The gray color means that the choice point is not fully explored. It is in fact completely unexplored.

Select the root by clicking on it, and press "n" (Next Solution, in the Search menu). This adds a green diamond[1], which corresponds to one solution. Double-clicking on the green diamond numbers the diamond (here it is 1) and displays the number and the solution in the Browser,

[1]The exact color depends on your screen; sometimes it is blue-green.

that is, `1#(nil#[1 2 3 4])`.

Now select the root again and press "a" (All Solutions, in the Search menu). This displays the tree in Figure 2. Each subtree's root node is a purple circle, which means that it is a fully explored choice point. Double-clicking on any node numbers the node and also displays what's known about the solution at that node. For example, double-clicking on node 2 displays `2#((1|2|_)#_)` and double-clicking on node 3 displays `3#([1]#[2 3 4])`.

4 Committed-Choice Logic Programming

A logic program in Oz can have multiple threads that bind shared variables. If predicates are defined only with **case**, **if**, and **cond**, then this is exactly committed-choice logic programming. The **case** and **if** statements are special cases of **cond**, which does a general don't-care choice, i.e., if the guard of any branch succeeds then execution can commit to that branch and discard all the others.

Here is a simple example of a producer-consumer program with flow control:

```
declare
proc {Producer N L}
   case L of X|Ls then X=N {Producer N+1 Ls}
   else skip end
end

fun {Consumer N L A}
   if N>0 then X L1 in L=X|L1 {Consumer N-1 L1 A+X}
   else A end
end
```

The producer generates a list `L` of consecutive increasing integers. The consumer sums the `N` first elements of `L`. The consumer asks for the next element by binding the list tail to a new list pair. The producer waits until the list is bound before generating the next element. Producer and consumer therefore run in lock step. A possible call is:

```
local L S in % Variable declaration
   thread {Producer 0 L} end
   thread S={Consumer 100000 L 0} end
   {Browse S}
end
```

The producer and consumer each runs in its own thread. The producer generates the list `[0 1 2 3 ...` and the consumer sums the list's first 100000 elements. The main thread immediately displays an unbound variable and later updates the display to `4999950000` when the consumer terminates.

Because only **`case`** and **`if`** are used, both the producer and the consumer have a precise logical semantics as well as an operational semantics. (This is not true for **`cond`** unless its conditions are mutually exclusive.) The **`if`** statement has the logical semantics $(c \wedge t) \vee (\neg c \wedge e)$ where c is the boolean condition, t is derived from the then part, and e is derived from the else part. The **`if`** statement has the following operational semantics. It waits until enough information exists to decide the truth or falsity of its boolean condition. At that point, it executes its then or else part.

5 Nondeterministic Concurrent Logic Programming

If a logic program has only a single thread and uses the **`dis`** statement to express nondeterminism, then its behavior is exactly like that of a Prolog program where the Prolog system is modified to do clause selection according to the Andorra principle. However, because of concurrency and first-class top levels, Oz lets you do much more. For example, let's say you have two sequential logic programs. There is a design choice when running them, i.e., whether to put them in the same top level or in different top levels:

- If the programs are independent, e.g., two independent queries to a database, then they should be run in different top levels. This ensures that each program gets a fair share of the processing power and that no wasted work is done.

- If the programs are dependent, i.e., they are cooperating to solve one problem, then it is often best to run them in the same top level. This ensures that they can share information. Fairness of each program is not important in this case. Rather, it is the progress made by *both programs considered together* that is important.

The second technique, dependent programs that cooperate, is not often used in logic programming, but it is very important for constraint programming. In Oz, a typical real-life constraint problem has hundreds, thousands, or even more active threads. Each thread observes the store and attempts to add information concurrently with the other threads. We call such a thread a *propagator* if it only adds correct information, i.e., it never creates a choice point [13, 15]. Propagators are implemented very efficiently in Oz and together with spaces they are the foundation of the Oz constraint programming model. Oz provides propagators for many complex constraints on the three constraint domains of finite domains, finite sets, and rational trees.

```
⟨S⟩  ::=  ⟨C⟩

       |  if ⟨C⟩ then ⟨S1⟩ else ⟨S2⟩ end

       |  case X
            of f1(l11:Y11  ...  l1m:Y1m) then ⟨B1⟩
            ...
            [] fn(ln1:Yn1  ...  lnm:Ynm) then ⟨Bn⟩
          else ⟨S⟩ end

       |  cond
             ⟨G1⟩ then ⟨B1⟩ [] ...  []  ⟨Gn⟩ then ⟨Bn⟩
          else ⟨S⟩ end

       |  dis
             ⟨G1⟩ then ⟨B1⟩ [] ...  []  ⟨Gn⟩ then ⟨Bn⟩
          end

       |  choice ⟨S1⟩ [] ...  []  ⟨Sn⟩ end

       |  ⟨Spaces⟩
```

Figure 3: Oz kernel support for logic programming.

6 Oz Kernel Support for Logic Programming

The full Oz language is defined in terms of a kernel language. The complete kernel language includes cells (explicit state), procedures, and threads in addition to logic programming support (see [19]). Figure 3 shows just the logic programming support. In this figure, ⟨C⟩ denotes a basic constraint, i.e., a constraint that is completely expressed in the store, ⟨G⟩, ⟨B⟩, and ⟨S⟩ denote statements (the first two are called guard and body), and ⟨Spaces⟩ denotes the support for computation spaces. Previous sections have explained part of the Oz support for logic programming, namely: (1) the don't-care disjunctive statements **`if`**, **`case`**, and **`cond`** (see Sections 2 and 4) and (2) the don't-know disjunctive statements **`dis`** and **`choice`** (see Section 3). The guards in the **`cond`** and **`dis`** statements can be arbitrary computations. If a guard is more than just a basic constraint, then we say that it is a *deep* guard. The then parts of a **`dis`** statement are optional. An omitted then part behaves as if it were "**`then skip`**", where **`skip`** is a statement that does nothing.

First-class top levels are not primitive. They are implemented in Oz through the *computation space* abstraction, which is outside the scope of this tutorial (see [11]). Computation spaces fully support deep guard execution, i.e., they can be nested to any level. Computation spaces interact with don't-know disjunctions to allow easy and efficient programming within Oz of arbitrary search strategies that work for arbitrary constraint domains. Most of the commonly-used search strategies are provided as library modules (see the `Search` module [2]). These modules include also some more unusual strategies, such as limited discrepancy search and saturation search, that are sometimes useful.

7 Constraints and Distribution

Up to now, we have explained how to write Prolog-style logic programs in Oz and how Oz extends what you can do in Prolog. But all this is just a warm-up exercise. Oz was never intended to be just a Prolog substitute. The main power of Oz is in constraint programming and distributed programming.

We summarize what the current Mozart release implements for distributed programming [6, 18]. Mozart completely separates the aspects of language semantics and distribution structure. The Oz semantics of all language entities are independent of their distribution structure. Furthermore, the network operations of the language entities are predictable, allowing efficient distributed applications to be written. These capabilities are implemented by means of a network layer that contains a distributed algorithm for each type of language entity, as well as distributed garbage collection [19, 5, 1]. Mozart has primitives for fully open computing, i.e., it is possible for independently-written applications that share no common ancestral information (such as an IDL definition) to connect and fruitfully interact. Finally, Mozart provides failure-detection primitives that allow building non-trivial fault tolerance abstractions within the language [17].

The constraint and distribution abilities of Oz can be combined. For example, the `Search` module implements a parallel search engine that is very useful for compute-intensive constraint problems [2]. The search engine is initialized by giving it a list of machine names and a script. The parallelism is completely transparent, i.e., the problem is specified without any knowledge of whether it is executed in parallel or not. The same script can be used with a top level, with the Explorer, and with a parallel search engine.

8 Conclusions and Perspectives

This tutorial gives an elementary introduction to doing logic programming in Oz. Along the way, we introduce first-class top levels, concurrency, and the Browser and Explorer tools. We explain how search-based and committed-choice logic programming with deep guards are integrated, and we outline how the logic programming support smoothly ties in to constraint programming.

Current active research topics in Oz include constraint programming for natural language processing, constraint debugging, fault tolerant and secure distributed execution, open computing architectures, and support for environments with limited computational resources.

Acknowledgements

The author thanks Denys Duchier, Seif Haridi, and Christian Schulte for their helpful comments on drafts of this paper. The author also thanks all the other contributors and developers of the Mozart system, of whose abilities this tutorial only gives the faintest echo. This research is partly financed by the Walloon Region of Belgium.

References

[1] Iliès Alouini and Peter Van Roy. Le protocole réparti de Distributed Oz (in French). In *Colloque Francophone sur l'Ingénierie des Protocoles (CFIP 99)*, pages 283–298, April 1999.

[2] Denys Duchier, Leif Kornstaedt, Tobias Müller, Christian Schulte, and Peter Van Roy. System modules. Technical report, 1999. In Mozart documentation, available at `http://www.mozart-oz.org`.

[3] Claire Gardent, Joachim Niehren, and Denys Duchier. Oz for natural language processing. Technical report, University of the Saarland, Saarbrücken, Germany, 1999.

[4] Seif Haridi and Nils Franzén. Tutorial of Oz. Technical report, 1999. In Mozart documentation, available at `http://www.mozart-oz.org`.

[5] Seif Haridi, Peter Van Roy, Per Brand, Michael Mehl, Ralf Scheidhauer, and Gert Smolka. Efficient logic variables for distributed computing. *ACM Transactions on Programming Languages and Systems*, 2000. To appear.

[6] Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3):223–261, May 1998.

[7] Joxan Jaffar and Michael Maher. Constraint logic programming: A survey. *J. Log. Prog.*, 19/20:503–581, May/July 1994.

[8] Mozart Consortium. The Mozart programming system (Oz 3), January 1999. Available at `http://www.mozart-oz.org`.

[9] Tobias Müller. Problem solving with finite set constraints in Oz. A tutorial. Technical report, 1999. In Mozart documentation, available at `http://www.mozart-oz.org`.

[10] Konstantin Popov. The Oz Browser. Technical report, 1999. In Mozart documentation, available at `http://www.mozart-oz.org`.

[11] Christian Schulte. Programming constraint inference engines. In Gert Smolka, editor, *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 519–533, Schloß Hagenberg, Austria, October 1997. Springer-Verlag.

[12] Christian Schulte. Oz Explorer–Visual constraint programming support. Technical report, 1999. In Mozart documentation, available at `http://www.mozart-oz.org`.

[13] Christian Schulte and Gert Smolka. Finite domain constraint programming in Oz. A tutorial. Technical report, 1999. In Mozart documentation, available at `http://www.mozart-oz.org`.

[14] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, September 1989.

[15] Gert Smolka. Problem solving with constraints and programming. *ACM Computing Surveys*, 28(4es), December 1996. Electronic Section.

[16] Leon Sterling and Ehud Shapiro. *The Art of Prolog–Advanced Programming Techniques*. Series in Logic Programming. MIT Press, 1986.

[17] Peter Van Roy. On the separation of aspects in distributed programming: Application to distribution structure and fault tolerance in Mozart. In *International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA 99)*, Tohoku University, Sendai, Japan, July 1999.

[18] Peter Van Roy, Seif Haridi, and Per Brand. Distributed programming in Mozart – A tutorial introduction. Technical report, 1999. In Mozart documentation, available at `http://www.mozart-oz.org`.

[19] Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. Mobile objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, September 1997.