Logic programming in the context of multiparadigm programming: the Oz experience *

Peter Van Roy[†], Per Brand[‡], Denys Duchier[§], Seif Haridi[¶], Martin Henz[‖] and Christian Schulte[**]

September 18, 2000

Abstract

Oz is a multiparadigm language that supports logic programming as one of its major paradigms. A multiparadigm language is designed to support different programming paradigms (logic, functional, constraint, object-oriented, sequential, concurrent, etc.) with equal ease. This article has two goals: to give a tutorial of logic programming in Oz and to show how logic programming fits naturally into the wider context of multiparadigm programming. Our experience shows that there are two classes of problems, which we call *algorithmic* and *search* problems, for which logic programming can help give practical solutions. *Algorithmic* problems have known efficient algorithms. *Search* problems do not have known efficient algorithms but can be solved with search. The Oz support for logic programming targets these two classes specifically, using the concepts needed for each. This is in contrast to the Prolog approach, which targets both classes with one set of concepts, which results in poor support for each class. We give examples that can be run interactively on the Mozart system, which implements Oz. To explain the essential difference between algorithmic and search programs, we define the Oz execution model. This model subsumes both concurrent logic programming (committed-choice-style) and search-based logic programming (Prolog-style). Furthermore, as consequences of its multiparadigm nature, the model supports new abilities such as first-class top levels, deep guards, active objects, and sophisticated control of the search process. Instead of Horn clause syntax, Oz has a simple, fully compositional, higher-order syntax that accommodates the abilities of the language. All this makes Oz one of the most advanced logic programming languages existing today. We give a brief history of Oz that traces the development of its main ideas and we summarize the lessons learned from this work. Finally, we give many entry points into the Oz literature.

1 Introduction

In our experience, logic programming can help give practical solutions to many different problems. We have found that all these problems can be divided into two classes, each of which uses a very different logic programming approach:

*This article is a much-extended version of the tutorial talk "Logic Programming in Oz with Mozart" given at the International Conference on Logic Programming, Las Cruces, New Mexico, Nov. 1999. Some knowledge of "traditional" logic programming (with Prolog or concurrent logic languages) is assumed.

[†] Université catholique de Louvain, B-1348 Louvain-la-Neuve, Belgium. Email: pvr@info.ucl.ac.be.

[‡] Swedish Institute of Computer Science, S-164 28 Sweden. Email: perbrand@sics.se.

[§] Universität des Saarlandes, D-66123 Germany. Email: Denys.Duchier@ps.uni-sb.de.

[¶] Swedish Institute of Computer Science, S-164 28 Sweden. Email: seif@sics.se.

[‖] National University of Singapore, Singapore 117543. Email: henz@comp.nus.edu.sg.

[**] Universität des Saarlandes, D-66123 Germany. Email: schulte@ps.uni-sb.de.

- **Algorithmic problems**. These are problems for which efficient algorithms are known. This includes simple parsing, rule-based expert systems, and transformations of complex symbolic data structures. For such problems, a logical specification of the algorithm is sometimes simpler than an imperative specification. In that case, *deterministic logic programming* or *concurrent logic programming* are natural ways to express it. Logical specifications are not always simpler. Sometimes an imperative specification is the best one can do, e.g., for problems in which state updating is frequent. Many graph algorithms are of the latter type.

- **Search problems**. These are problems for which efficient algorithms are not known. This may be either because such algorithms are not possible in principle, e.g., NP-complete problems [22], or because the algorithms have not been made explicit, e.g., problems with very complex specifications. Some examples of search problems are optimization problems (planning, scheduling, configuration), natural language parsing, and theorem proving. These kinds of problems can be solved by doing search, i.e., with *nondeterministic logic programming*. But search is a dangerous tool. If used naively, it does not scale up to real applications. For a real application, all effort must be done to reduce the need for search: use strong (global) constraints, concurrency for cooperative constraints, heuristics for the search tree, etc. For problems with complex specifications, using sufficiently strong constraints sometimes results in a polynomial-time algorithm (see, e.g., [40]).

The Oz support for logic programming is targeted specifically towards algorithmic and search problems. The first part of this article (Sections 2–6) shows how to write logic programs in Oz for these problems. Section 2 introduces deterministic logic programming, which targets algorithmic problems. It is the most simple and direct way of doing logic programming in Oz. Section 3 shows how to do nondeterministic logic programming in the Prolog style. This targets neither algorithmic nor search problems, and is therefore only of pedagogical interest. Section 4 shows how to do concurrent logic programming in the classical tradition. This targets more algorithmic problems. Section 5 extends Section 4 with explicit state. In our experience, explicit state is essential for practical concurrent logic programming. Section 6 expands on Section 3 to show how search can be made practical.

The second part of this article (Sections 7–9) focuses on the essential difference between the techniques used to solve algorithmic and search problems. This leads to the wider context of *multiparadigm* programming. Section 7 introduces the Oz execution model, which has a strict functional core and extensions for explicit state, concurrency, search, and lazy evaluation. The section explains how these extensions can be used in different combinations to provide different programming paradigms. In particular, Section 7.4 explains the abstraction of *computation spaces*, which is the main tool for doing search in Oz. Spaces make possible a deep synthesis of concurrent and constraint logic programming. Section 8 gives a short history of Oz, whose roots lie in concurrent constraint programming. Finally, Section 9 summarizes the lessons we have learned in the Oz project on how to do practical logic programming and multiparadigm programming.

2 Deterministic logic programming

We call *deterministic* logic programming the case when the algorithm's control flow is completely known and specified by the programmer. No search is needed. This is perfectly adapted to sequential algorithmic problems. For example, a deterministic naive reverse can be written as follows in Oz:

```
proc {Append Xs Ys Zs}
   case Xs
   of nil then Ys=Zs
```

```
      [] X|Xr then Zr in
         Zs=X|Zr {Append Xr Ys Zr}
      end
   end

   proc {NRev Xs Ys}
      case Xs
      of nil then Ys=nil
      [] X|Xr then R in
         {NRev Xr R}
         {Append R [X] Ys}
      end
   end
```

This syntax should be vaguely familiar to people with some knowledge of Prolog and functional programming [80, 45, 82, 15]. We explain briefly its semantics to highlight the relationship to logic programming. In this code, all capitalized identifiers are logic variables, which have to be declared explicitly. The "`Zr` **in**" and "`R` **in**" explicitly declare `Zr` and `R`, i.e., they are existentially quantified with an explicit local scope. Lists are either the atom `nil` or a pair of an element `X` and a list `Xr`, written `X|Xr`. The **case** statement consists of clauses separated by "`[]`". Each clause has a pattern and a clause body. The first clause that matches the pattern is chosen. Variables in a pattern are implicitly declared and are local to its clause. For example, `X|Xr` declares `X` and `Xr`.

Since the direction of control flow is known, the above definitions can be written in a more functional style:

```
fun {Append Xs Ys}
   case Xs
   of nil then Ys
   [] X|Xr then X|{Append Xr Ys}
   end
end

fun {NRev Xs}
   case Xs
   of nil then nil
   [] X|Xr then {Append {NRev Xr} [X]}
   end
end
```

This is just syntactic sugar; in Oz, a function is just a shorter way of writing a procedure where the procedure's last argument is the function's output. The statement `Ys={NRev Xs}` has identical behavior to the procedure call `{NRev Xs Ys}`. The function can be called as follows:

```
{Browse {NRev [1 2 3 4 5]}}
```

where `Browse` opens a concurrent graphic tool, the Browser, to display the result [58].

The `Append` and `NRev` calls do not do search. If there is not enough information to continue, then the computation may suspend. For example, take these two calls:

```
declare X Y in
{Browse {Append [1] X}}
{Browse {Append Y [2]}}
```

The first call, `{Append [1] X}`, will run successfully since `Append` does not need the value of its second argument. It will display `1|X`. The second call, `{Append Y [2]}`, will suspend the thread it is executing in. Nothing is displayed. If another thread binds `Y`, then execution will continue. Programming errors often result in suspensions; when this happens the Oz Debugger is a useful tool to find out what is suspending [43].

This is how Oz supports deterministic logic programming. Syntactically, it looks like functional programming with dynamic typing (see Section 9.5) [1]. In particular, one can use higher-order procedures to modularize programs. However, this programming style extends functional programming because logic variables can be unbound. This allows powerful programming techniques based on incomplete data structures, such as difference lists [80]. To be precise, in this style all calculations are done over a constraint store: unifications add information to the store and conditionals (**if** and **case**) wait for the store to entail given information. This is purely declarative logic programming with an operational semantics that is fully specified and deterministic.

3 Nondeterministic logic programming

We call *nondeterministic* logic programming the situation when *search* is used to provide completeness. Using search allows finding solutions when the algorithm is not completely known. Oz provides a disjunction, called **dis** statement, as a simple way to introduce search. The **dis** statement is determinacy-directed. That is, if all branches of the disjunction fail except for one, then that remaining branch is taken. If more than one branch remains, then a choice point is created. The **dis** statement extends Prolog-style execution to interleave deterministic and non-deterministic reduction. This extension was discovered by D.H.D. Warren and called the *Andorra principle* [23, 60].

The **dis** statement is a powerful way to do Prolog-style generative execution. However, the **dis** statement does not scale up to real-world search problems.[1] In our opinion, its primary values are pedagogical and exploratory. That is, it can be used on small examples to explore and understand a search space, but it will not work on large examples due to search space explosion. See Section 6 for a scalable way to do search.

Here's a nondeterministic naive reverse with **dis**:

```
proc {Append Xs Ys Zs}
   dis           Xs=nil          then Ys=Zs
   [] X Xr Zr in Xs=X|Xr Zs=X|Zr then {Append Xr Ys Zr}
   end
end

proc {NRev Xs Ys}
   dis        Xs=nil  then Ys=nil
   [] X Xr in Xs=X|Xr then {Append {Nrev Xr} [X] Ys}
   end
end
```

Each clause of the **dis** statement has a guard and a body. Explicit variable declarations are needed in the guards because of full compositionality, i.e., all statements (including **dis**) can be arbitrarily nested. The **dis** statement supports deep guards, which is useful when the actual guard needs to be obtained at run-time through calculation.

This program can be called in many ways, e.g., by lazy depth-first search (similar to a Prolog top level)[2], eager search, or interactive search (with the Explorer tool [67, 87]). All of these search abilities are programmed in Oz using the notion of computation space (see Section 7). In most cases, the programmer will never use spaces directly (although he can), but will use one of the many predefined search abstractions provided in the `Search` module (see Section 6.2).

The simplest way to use search in Oz is by using the *search object*. The search object provides a lazy depth-first search, similar to a Prolog top level. Here's how to use it:

[1]For problems with a small search space, **dis** may be sufficient. For example, a practical diagnostics generator for the VLSI-BAM microprocessor was written in Prolog [31].

[2]Note that lazy search is orthogonal to lazy evaluation (see Section 7).

```
declare
% Define a new search query:
proc {P S} X#Y=S in {Append X Y [1 2 3 4 5]} end

% Set up a new search engine:
E={New Search.object script(P)}

% Get and display the first solution: (and others, when repeated)
{Browse {E next($)}}
```

This example introduces an important syntactic short-cut: the use of "`$`" (dollar) as a nesting marker. The dollar indicates the part of a nested expression that is passed to the enclosing statement. That is, the statement `{Browse {E next($)}}` has identical behavior to:

```
local X in
   {E next(X)}
   {Browse X}
end
```

The first call of the browser displays the first solution as `[nil#[1 2 3 4 5]]`, that is, a one-element list containing a solution. The solution is given as a single term, e.g., `X#Y`, by using the pair operator "`#`". Successive calls display the solutions `[[1]#[2 3 4 5]]`, ..., `[[1 2 3 4 5]#nil]`. When there are no more solutions, then `nil` is displayed instead of a one-element list.

The standard Oz approach is to use search only for problems that require it. For algorithmic problems, one doesn't need to learn about search. This is unlike Prolog, where search is ubiquitous: even procedure application is defined in terms of resolution, and thus search. In Oz, the **`dis`** statement explicitly creates a choice point, and search abstractions (such as `Search.object`, above) allow it to be controlled. However, **`dis`** by itself is a bit too simplistic, since the choice point is statically placed. The usual way to add choice points in Oz is with abstractions that dynamically create a choice point depending on the state of the computation. The heuristics used are called the *distribution strategy*. For example, the procedure `FD.distribute` allows to specify the distribution strategy for problems using finite domain constraints. Section 6.3 gives an example of this approach.

4 Concurrent logic programming

In *concurrent* logic programming, programs are written as a set of don't-care predicates and executed concurrently. That is, at most one clause is chosen from each predicate invocation, in an indeterministic way from all the clauses whose guards are true. This style of logic programming is incomplete, just like deterministic logic programming. Only a small part of the search space is explored due to the guarded clause selection. The advantage is that programs are concurrent, and concurrency is essential for programs that interact with their environment, e.g., for agents, GUI programming, OS interaction, etc. Many algorithmic problems are of this type. Concurrency also permits a program to be organized into parts that execute independently and interact only when needed. This is an important software engineering property.

In this section, we show how to do concurrent logic programming in Oz. In fact, the full Oz language allows concurrency and search to be used together (see Section 7). The clean integration of both in a single language is one of the major strengths of Oz. The integration was first achieved in Oz's immediate ancestor, AKL, in 1990 [24]. Oz provides it in a much simpler and more expressive way. Oz can be seen as a compositional and higher-order version of AKL.

4.1 Implicit versus explicit concurrency

In early concurrent logic programming systems, concurrency was implicit, driven solely by data dependencies [73]. Each body goal implicitly ran in its own thread. The hope was that this would make parallel execution easy. But this hope has not been realized, for several reasons. The overhead of implicit concurrency is too high, parallelism is limited without rewriting programs, and reasoning about programs (e.g., termination) is difficult. To reduce the overhead, it is possible to do lazy thread creation, that is, to create a new thread only when the parent thread would block. This approach has a nice slogan, "as sequential as possible, as concurrent as necessary," and it allows an efficient implementation. But the approach is still inadequate because reasoning about programs remains hard.

After implementing and experimenting with both implicit concurrency and lazy thread creation, the current Oz decision is to do only explicit thread creation (see Section 8). We do not miss the dubious magic of the other two approaches. Explicit thread creation simplifies debugging and reasoning about programs, and is efficient. Furthermore, experience shows that parallelism (i.e., speedup) is not harder to obtain than before; it is still the programmer's responsibility to know what parts of his program can potentially be run in parallel.

4.2 Concurrent producer-consumer

A classic example of concurrent logic programming is the asynchronous producer-consumer. The following program asynchronously generates a stream of integers and sums them. A *stream* is a potentially unbounded list that may be incomplete, i.e., whose tail may be an unbound logic variable.

```
fun {Generate N Limit}
   if N<Limit then
      N|{Generate N+1 Limit}
   else nil end
end

fun {Sum Xs A}
   case Xs
   of X|Xr then {Sum Xr A+X}
   [] nil then A
   end
end

local Xs S in
   thread Xs={Generate 0 150000} end  % Producer thread
   thread S={Sum Xs 0} end            % Consumer thread
   {Browse S}
end
```

This executes as expected in the concurrent logic programming framework. The producer, `Generate`, and the consumer, `Sum`, run in their own threads. They communicate through the shared variable `Xs`, which is a stream of integers. The **case** statement in `Sum` synchronizes on `Xs` being bound to a value.

4.3 Lazy producer-consumer

In the above producer-consumer example, it is the producer that decides how many list elements to generate. This is called *supply-driven* or *eager* execution. This is an efficient technique if the total amount of work is finite and does not use many system resources (e.g., memory or calculation time). On the other hand, if the total work potentially uses many resources, then it may be better to use *demand-driven* or *lazy* execution. With lazy execution, the consumer decides how many list elements to generate. If an extremely large or a potentially unbounded number of list elements are needed, then lazy execution will use many fewer system resources at any given point in time. Problems that are impractical with eager execution can become practical with lazy execution.

Lazy execution can be implemented in two ways in Oz. The first way, which is applicable to any language, is to use *triggers*. The producer and consumer are modified so that the consumer asks the producer for additional list elements. In our example, the simplest way is to use logic variables as triggers. The consumer binds the end of stream to `X|Xr`. The producer waits for this and binds `X` to the next list element.

Programming triggers is cumbersome because it requires the producer to accept explicit communications from the consumer. A better way is for the language to support laziness directly. That is, the language semantics would ensure that a function is evaluated only if its result were needed. Oz supports this by annotating the function as "`lazy`". Here's how to do the previous example with a lazy function that generates a potentially infinite list:

```
fun lazy {Generate N}
   N|{Generate N+1}
end

fun {Sum Xs A Limit}
   if Limit>0 then
      case Xs
      of X|Xr then
         {Sum Xr A+X Limit-1}
      end
   else A end
end

local Xs S in
   thread Xs={Generate 0} end
   thread S={Sum Xs 0 150000} end
   {Browse S}
end
```

Here the consumer, `Sum`, decides how many list elements should be generated. The addition `A+X` implicitly triggers the generation of a new list element `X`. Lazy execution is part of the Oz execution model; Section 7.2 explains how it works.

4.4 Coroutining

In our experience, a system with efficient preemptive concurrency does not need to support coroutining, i.e., nonpreemptive concurrency. Sequential systems often support coroutining as a simple way to get some of the abilities of concurrency. For example, most modern Prolog systems support coroutining, but in a way that cannot be used together with backtracking search. The coroutining is either supported directly, as in IC-Prolog [12, 13], or indirectly, by means of an operation called `freeze`, which provides data-driven computation. The `freeze(X,G)` operation, sometimes called `geler(X,G)` from Prolog II which pioneered it [14], sets up the system to invoke the goal `G` when the variable `X` is bound [80]. With `freeze` it is possible to have "nonpreemptive threads" that explicitly hand over control to each other by binding variables. Because Prolog's search is based on global backtracking, the "threads" are not independent: if a thread backtracks, then other threads may be forced to backtrack as well. Prolog programming techniques that depend on backtracking, such as search, deep conditionals, and exceptions, cannot be used if the program has to switch between threads.

5 State

From a theoretical point of view, state has often been considered a forbidden fruit in logic programming. In our experience, this attitude is wrong. Outlawing state is depriving oneself of a useful tool. What's more, state cannot be so easily eliminated: it always comes back in one way or another, like the proverbial cat. Functional programmers have also learned this lesson, and have integrated state into functional languages, e.g., by means of `set` operations in LISP/Scheme [79, 1], references in ML [52], and monads in Haskell [93]. In logic programming, state must be faced as well. This issue has been addressed within the context of Prolog. In addition to the heavy machinery of assert/retract, there exist tools to use state in Prolog while keeping a semblance of logical semantics. See for example SICStus Objects [8], Prolog++ [53], and the Logical State Threads package [38]. An ancestor of the latter was used to help write the Aquarius Prolog compiler [86, 89].

5.1 Explicit state creation

In Oz, as in its ancestor AKL, state is an explicit part of the basic execution model. There are many equivalent ways to add state to the basic model. One way, which was pioneered by AKL, is to add the concept of a *port*, which is a pair of a name `P` and a stream `Xs` [37]. Ports exist in Oz as well. They have just two operations:

```
{NewPort Xs P}   % Create new port with name P and stream Xs
{Send P X}       % Add X to port's stream, i.e., Xs=X|Xr
```

Each `Send` asynchronously adds one more element to the port's stream. The port keeps an internal reference to the stream's unbound tail. Repeated sends in the same thread cause the elements to appear in the same order as the sends. There are no other ordering constraints on the stream.

Using ports gives us the ability to have named active objects. An *active object*, in its simplest form, pairs an object with a thread. The thread reads a stream of internal and external messages, and invokes the object for each message. The Erlang language is based on this idea [6]. Erlang extends it by adding to each object a mailbox that does retrieval by pattern matching.

Oz has a full-featured concurrent object system which is completely defined in terms of *cells*, the Oz version of state [28, 27]. A cell is a kind of mutable pointer. The object system includes multiple inheritance, fine-grained method access control, and first-class messages. See Sections 7.1 and 7.3 for more on cells and how they underlie the object system. While AKL objects are active objects that encapsulate a thread, the Oz object system has passive objects with separate threads. The latter approach is particularly expressive and efficient [51, 76]. From a theoretical point of view, ports and cells are equivalent: each can express the other in a simple way.

From the perspective of concurrent logic programming, explicit state amounts to the addition of a constant-time n-way stream merge. That is, any number of threads can concurrently send to the same port, and each send will take constant time. Without explicit state this is impossible if n is known only at run-time; one instead has to build a tree of stream mergers. With n senders, this multiplies the message sending time by $O(\log n)$. We know of no simple way to solve this problem other than by adding explicit state to the execution model.

5.2 Creating an active object

Here's an example that uses a port to make an active object:

```
proc {DisplayStream Xs}
   case Xs of X|Xr then {Browse X} {DisplayStream Xr}
   else skip end
end

declare P in     % P has global scope
local Xs in      % Xs has local scope
   {NewPort Xs P}
```

```
    thread {DisplayStream Xs} end
end
```

Sending to `P` sends to the active object. Any number of clients can send to the active object concurrently:

```
thread {Send P 1} {Send P 2} ... end   % Client 1
thread {Send P a} {Send P b} ... end   % Client 2
```

The elements `1`, `2`, `a`, `b`, etc., will appear fairly on the stream `Xs`. Port fairness is guaranteed because of thread fairness in the Mozart implementation.

Here is a more compact way to define the active object's thread:

```
thread
   {ForAll Xs proc {$ X} {Browse X} end}
end
```

where `ForAll` is a high-order procedure that applies a unary procedure to all elements of a list. `ForAll` keeps the dataflow synchronization when traversing the list. This is an example how higher-orderness can be used to modularize a program: the iteration is separated from the action to be performed on each iteration.

6 More on search

We have already introduced search in Section 3 by means of the **dis** statement and the lazy depth-first search abstraction `Search.object`. This tells only part of the story, since **dis** is impractical for most real programs. In this section we show how to make search practical. This section only scratches the surface of how to use search in Oz; for more information we suggest the Finite Domain and Finite Set tutorials in the Mozart system documentation [70, 55].

6.1 Aggregate search

One of the powerful features of Prolog is its ability to generate aggregates based on complex queries, through the built-in operations `setof/3` and `bagof/3`. These are easy to do in Oz; they are just special cases of search abstractions. In this section we show how to implement `bagof/3`. Consider the following small biblical database (taken from [80]):

```
proc {Father F C}
   dis F=terach  C=abraham
    [] F=terach  C=nachor
    [] F=terach  C=haran
    [] F=abraham C=isaac
    [] F=haran   C=lot
    [] F=haran   C=milcah
    [] F=haran   C=yiscah
   end
end
```

The Prolog predicate:

```
children1(X, Kids) :- bagof(K, father(X,K), Kids).
```

(without existential quantification) is defined in Oz as follows:

```
proc {Children1 X Kids}
   {Search.base.all proc {$ K} {Father X K} end Kids}
end
```

The `Search.base.all` abstraction takes a unary procedure and returns the list of all solutions to the procedure. The `Children1` definition is deterministic; it assumes `X` is known and it returns `Kids`. To search over different values of `X` the following definition should be used instead:

```
proc {Children1 X Kids}
   {Father X _}
   {Search.base.all proc {$ K} {Father X K} end Kids}
end
```

The call `{Father X _}` creates a choice point on `X`. The "_" is syntactic sugar for **`local`** `X` **`in`** `X` **`end`**, which is just a new variable with a very small scope.
The Prolog predicate:

```
children2(Kids) :- bagof(K, X^father(X,K), Kids).
```

(with existential quantification) is defined in Oz as follows:

```
proc {Children2 Kids}
   {Search.base.all proc {$ K} {Father _ K} end Kids}
end
```

The Oz solution uses _ to add a new existentially-scoped variable. The Prolog solution, on the other hand, introduces a new concept, namely the "existential quantifier" `X^`, which only has meaning in terms of `setof/3` and `bagof/3`. The fact that this notation denotes an existential quantifier is arbitrary. The Oz solution introduces no new concepts. It really does existential quantification inside the search query.

6.2 Simple search procedures

The procedure `Search.base.all` shown in the previous section is just one of a whole set of search procedures provided by Oz for elementary nondeterministic logic programming. We give a short overview; for more information see the System Modules documentation in the Mozart system [19]. All procedures take as argument a unary procedure `{P X}`, where `X` stands for a solution to be generated. Except for lazy search, they all provide depth-first search (one and all solution) and branch-and-bound search (with an order function). Here are the procedures:

- **Basic search**. This is the simplest to use; no extra parameters are needed.
- **General-purpose search**. This allows parametrizing the search with the maximal recomputation distance (for optimizing time and memory use), with an asynchronous kill procedure to allow stopping infinite searches, and with the option to return solutions either directly or encapsulated in computation spaces (see Section 7.4). Search implemented with spaces is competitive in time and memory with systems using trailing [66]. Using encapsulation, general-purpose search can be used as a primitive to build more sophisticated searches.
- **Parallel search**. When provided with a list of machines, this will spread out the search process over these machines transparently. We have benchmarked realistic constraint problems on up to six machines and we see linear speedups [68].
- **Lazy search**. This provides next solution and last solution operations, a stop operation, and a close operation. This is a first-class Prolog top level.
- **Explorer search**. The Explorer is a concurrent graphic tool that allows to visualize and interactively guide the search process [67, 64]. It is invaluable for search debugging and for gaining understanding of the structure of the problem.

All of these procedures are implemented in Oz using computation spaces (see Section 7.4). Many more specialized search procedures are available for constraint programming, and the user can easily define his own.

6.3 The scalable way to do search

The original motivation for doing search in Oz comes from constraint programming. Oz uses the following two-step approach to do search, which is a concurrent version of the approach used in (sequential) constraint logic programming:

- First, declaratively specify the problem by means of constraints. The constraints have an operational as well as a declarative reading. The operational reading specifies the deductions that the constraints can make locally. To get good results, the constraints must be able to do nontrivial deductions over big parts of the problem. Such constraints are called "global".
- Second, define and explore the search tree in a controlled way, using heuristics to exploit the problem structure. The general technique is called "propagate and distribute", because it alternates propagation steps (where the constraints propagate information amongst themselves) with distribution steps (where a choice is selected in a choice point) [75].

This approach is widely applicable. For example, it is being applied successfully to natural language processing [17, 40, 18]. In this section, we show how to solve a simple integer puzzle. Consider the problem of finding nine distinct digits A, B, ..., I, so that the following equation holds:

$$A/BC + D/EF + G/HI = 1$$

As a constraint problem, this can be specified as follows:

```
functor Fractions  % Name of module specification
   import FD       % Needs the module FD
   export Script   % Script defines the problem
define
   proc {Script Sol}
      sol(a:A b:B c:C d:D e:E f:F g:G h:H i:I)=Sol
      BC={FD.decl} EF={FD.decl} HI={FD.decl}
   in
      %%% The constraints:
      Sol:::1#9          % Each letter represents a digit
      {FD.distinct Sol}  % All digits are different
      BC=:10*B+C         % Definition of BC
      EF=:10*E+F         % Definition of EF
      HI=:10*H+I         % Definition of HI
      A*EF*HI+D*BC*HI+G*BC*EF=:BC*EF*HI  % Main constraint
      %%% The search heuristic: (shape of search tree)
      {FD.distribute ff Sol}
   end
end
```

The unary procedure `{Script Sol}` fully defines the problem and the shape of the search tree. The problem is specified as a conjunction of constraints on `Sol`, which represents the solution. The constraints are expressed in terms of *finite domains*, i.e., finite sets of integers. For example, the notation `1#9` represents the set $\{1, 2, ..., 9\}$. The constraints are defined in the module `FD` [19]. For example, `FD.distinct` is a global constraint that asserts that all its component variables are distinct integers.

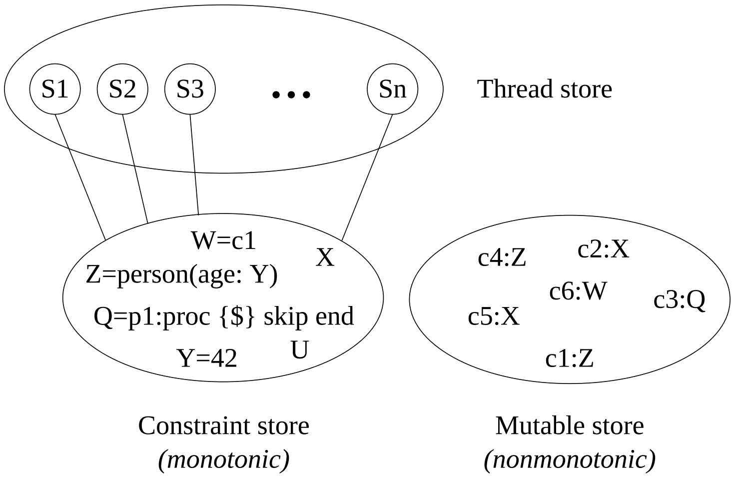

Figure 1: The Oz store

`Fractions` defines `Script` inside a *functor*, i.e., a module specification, in Oz parlance. The functor defines explicitly what process-specific resources the module needs. This allows us to set up a parallel search engine that spreads the constraint solving over several machines [20]. (If execution is always local, then the functor is not needed and it is enough to define the procedure `Script`.) Let's set up a parallel search engine:

```
E={New Search.parallel
       init(adventure:1#rsh galley:1#rsh norge:1#rsh)}
```

This sets up an engine on three machines, namely `adventure`, `galley`, and `norge`. The engine is implemented using computation spaces (see Section 7.4) and Mozart's support for distributed computing (see [26]). A single process is created on each of these machines using the `rsh` operation (other operations are possible including `ssh` for secure communication and `sh` for shared-memory multiprocessors). The following command does parallel search on the problem specified in `Fractions`:

```
{Browse {E all(Fractions $)}}
```

This installs the functor `Fractions` and generates all the solutions. This is an example of the scalable way to do search: use global constraints and search heuristics, and then use parallel execution if necessary for performance.

Oz is currently one of the most advanced languages for programming search. Competitors are CLAIRE and SaLSA [9, 41, 10] and OPL [85]. Search is also an important part of constraint programming in general [46].

7 The Oz execution model

So far, we have highlighted different parts of Oz without showing how they interact, something like the mythical elephant that is all things to all people. This section gives the simple execution model that underlies it all. We define the execution model in terms of a *store* (Section 7.1) and a *kernel language* (Section 7.2). Section 7.3 explains how different subsets of the kernel language support different programming paradigms. The section also explains why supporting multiple paradigms is useful. Finally, Section 7.4 defines computation spaces and how they are used to program search.

```
⟨S⟩ ::= skip                                    | ⟨S1⟩ ⟨S2⟩          CORE
      | proc {P X} ⟨S⟩ end                      | {P X}
      | try ⟨S1⟩ catch X then ⟨S2⟩ end          | raise X end
      | X= f(l1:Y1 ... lm:Ym)                   | X=Y
      | local X in ⟨S⟩ end
      | if X then ⟨S1⟩ else ⟨S2⟩ end
      | case X of
          Y1a ... Y1z in f1(l11:Y11 ... l1m:Y1m) then ⟨S1⟩
          [] ... []
          Yna ... Ynz in fn(ln1:Yn1 ... lnm:Ynm) then ⟨Sn⟩
        else ⟨S⟩ end

      | {NewCell X C}                                                STATE
      | {Exchange C X Y}

      | thread ⟨S⟩ end                                               CONCURRENCY
      | cond
          Y1a ... Y1z in ⟨G1⟩ then ⟨S1⟩
          [] ... []
          Yna ... Ynz in ⟨Gn⟩ then ⟨Sn⟩
        else ⟨S⟩ end

      | ⟨Space⟩                                                      SEARCH

      | {ByNeed P X}                                                 LAZINESS
```

Figure 2: The Oz kernel language

7.1 The store

The Oz store consists of three parts (see Figure 1): a constraint store, a mutable store, and a thread store. The constraint store contains equality constraints over the domain of rational trees (see also Section 7.4). In other words, this store contains logic variables that are either unbound or bound. A bound variable references a term (i.e., atom, record, procedure, or name) whose arguments themselves may be bound or unbound. Unbound variables can be bound to unbound variables, in which case they become identical references. The constraint store is *monotonic*, i.e., bindings can only be added, not removed or changed.

The mutable store consists of mutable pointers that contain references into the constraint store. Mutable pointers are also called *cells* [91]. A mutable pointer consists of two parts: its *name*, which is a value, and its *content*, which is a reference into the constraint store. The mutable store is *nonmonotonic* because pointer contents can be changed.

The thread store consists of a set of threads. Each thread is defined by a statement S_i. The only way for threads to communicate and synchronize is by means of shared references into the constraint store. We say a thread is *runnable*, also called *ready*, if it can execute its first statement. Threads are *dataflow* threads, i.e., a thread becomes runnable when the first statement's needed arguments are bound. If a needed argument is unbound then the thread automatically blocks until the argument is bound. Since the constraint store is monotonic, a thread that is runnable will stay runnable at least until it executes its first statement. The system guarantees weak fairness, which implies that a runnable thread will eventually execute its first statement.

7.2 The kernel language

All Oz execution can be defined in terms of a simple kernel language, whose syntax is defined in Figure 2. The full Oz language provides syntactic support for additional language entities (such as functions, ports, objects, classes, and functors). The system hides their efficient implementation while respecting their definitions in terms of the kernel language. From the kernel language viewpoint, n-ary functions are just $(n+1)$-ary procedures, where the last argument is the function's output. In Figure 2, statements are denoted by $\langle \mathsf{S} \rangle$ and $\langle \mathsf{G} \rangle$, space operations by $\langle \mathsf{Space} \rangle$ (see Figure 4), and logic variables by italic capitals such as X and Y. The variable P references a procedure and the variable C references a cell.

The semantics of the kernel language are given in [74], [68], [91], and [25]. Some syntactic sugar is added to Figure 2 to make it more intuitive. In particular, Figure 2 includes three redundant statements, **`local`**, **`if`**, and **`case`**, which can all be expressed with **`cond`**. The kernel language splits naturally into five parts:

- *CORE*: The core is strict functional programming over a constraint store. This is exactly deterministic logic programming with explicit sequential control. The **`if`** statement expects a boolean argument (**`true`** or **`false`**). The **`case`** statement does pattern matching. New variables can be declared in each clause of the **`case`**, e.g., the variables $\{\texttt{Y}_{1a}, ..., \texttt{Y}_{1z}\}$ are declared with a scope that covers the first clause.

- *STATE*: The state support adds explicit cell creation and an exchange operation, which atomically reads a cell's content and replaces it with a new content. This is deceptively simple, yet it has vast ramifications for practical concurrent logic programming and object-oriented programming.

- *CONCURRENCY*: The concurrency support adds explicit thread creation and the **`cond`** statement, which provides don't-care nondeterminism (committed choice to one alternative). We distinguish *flat* **`cond`**, in which all guards are equality constraints, and *deep* **`cond`**, in which this is not so. As in the **`case`** statement, new variables can be declared in each clause of the **`cond`**. The **`cond`** statement is the logical form of a Dijkstra guarded command, which allows to wait concurrently on more than one event. An interesting fact is that deep **`cond`** can be programmed with flat **`cond`** and computation spaces (see Section 7.4.5).

- *SEARCH*: The search support adds operations on computation spaces (shown as $\langle \mathsf{Space} \rangle$, see Section 7.4). A computation space encapsulates a choice point, i.e., don't-know nondeterminism, allowing the program to decide how to pick alternatives. Section 7.4 explains spaces in more detail and shows how to program search with them. Computation spaces can be used in many ways to specify choice points. For example, the **`dis`** statement, which is used in the examples of Sections 3 and 6.1, can be programmed with flat **`cond`** and spaces (see Section 7.4.5).

- *LAZINESS*: The laziness support adds the `ByNeed` function, which allows to express lazy evaluation as provided by functional languages such as Haskell [50, 48, 32].[3] Calling `X={ByNeed F}` returns a read-only variable `X`. Any attempt to bind `X` will suspend the attempting thread. When the value of `X` is needed, then the function `{F}` is executed in a new thread. When `F` is fully reduced, then its result is bound to `X`.

The `lazy` annotation used in Section 4.3 is defined in terms of `ByNeed`. For example, the function:

```
fun lazy {Generate N}
   N|{Generate N+1}
end
```

[3] In Mozart 1.1.0, the module `Value` contains this operation: `ByNeed=Value.byNeed`.

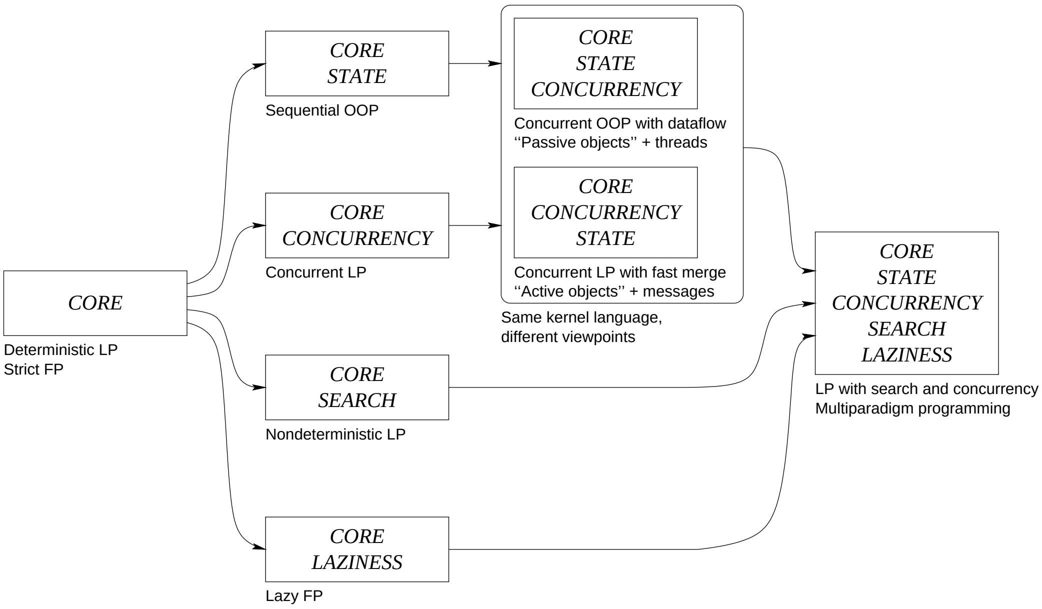

Figure 3: Programming paradigms in Oz

is defined as:

```
fun {Generate N}
   fun {F} N|{Generate N+1} end
in
   {ByNeed F}
end
```

Thanks to `ByNeed`, `F` will only be called when the value of `{Generate N}` is needed. Since the `lazy` annotation is given explicitly by the programmer, its implementation in Oz is less important for overall efficiency than in languages with implicit laziness such as Haskell [32]. Oz threads are cheap enough that the above semantic definition is practical.

7.3 Multiparadigm programming

Many different programming styles or "paradigms" are possible by limiting oneself to different subsets of the kernel language. Some popular styles are object-oriented programming (programming with state, encapsulation, and inheritance), functional programming (programming with values and pure functions), constraint programming (programming with partial information), and sequential programming (programming with a totally-ordered sequence of instructions). The full Oz language provides syntactic and implementation support that makes these paradigms and many others equally easy to use. The execution model is simple and general, which allows the different styles to coexist comfortably. This ability is known as *multiparadigm* programming.

An interesting early discussion on multiparadigm programming in Oz is given in [47]. This paper gives examples in logic, functional, and object-oriented styles. The paper also has an in-depth discussion of Prolog and a survey of related work as of 1995.

The justification of limiting oneself to one particular paradigm is that the program may be easier to write or reason about. For example, if the **thread** construct is not used, then the program is purely sequential. If the `ByNeed` operation is not used, then the program is strict. Experience shows that different levels of abstraction often need different paradigms [63].

Even if the same basic functionality is provided, it may be useful to view it according to different paradigms depending on the application needs [42]. Some interesting subsets of the kernel language are shown in Figure 3:

- The core by itself provides deterministic logic programming and strict functional programming (see Section 2).
- The core with state provides sequential object-orientation. This case is especially interesting; it is summarized below. For more information please see [74, 28, 27].
- The core with concurrency provides concurrent logic programming, in the committed-choice style (see Section 4).
- The core with search provides nondeterministic logic programming (see Sections 3 and 6).
- The core with laziness provides lazy functional programming. Lazy evaluation is the ability to trigger calculations based on need, which is a useful form of automatic resource management.
- The core with concurrency and state provides concurrent logic programming together with active objects (see Section 5). Doing the construction in the opposite order, i.e., taking the core with state and then adding concurrency, provides passive objects and threads. These two cases have identical expressive power; they are just syntactic variants with different performance trade-offs [42].
- Finally, the full model, core with state, concurrency, search, and laziness, gives logic programming with both search and concurrency. It is the most flexible, powerful, and practical model for logic programming existing today.

The full model gives the advantages of multiparadigm programming while keeping the language simple. How is it possible for such a simple language to support such different programming styles? There is no miracle, but just the careful study of what characterizes each paradigm.[4] In the full model, the constructs of the different paradigms don't just sit beside one another and make life hard! As Figures 2 and 3 show, the total number of different constructs is much less than the sum of the number of constructs in each paradigm. This is because paradigms have many constructs in common. Sometimes a construct of one paradigm is a special case of a construct from another paradigm (like **`if`**, which is a special case of **`cond`**). Another possibility is that a construct of one paradigm can be built out of more primitive constructs. A good example of this is sequential object-oriented programming, which can be built from the core with state (see [74] for details):

- Procedures behave as objects when they internally reference state.
- Methods are different procedures that reference the *same* state.
- Classes are records that group related method definitions.
- Inheritance is an operation that takes a set of method definitions and one or more class records, and constructs a new class record.
- Creation of new object instances is done by a higher-order procedure that takes a class record and associates a new state pointer with it.

[4] And the non-dogmatic acceptance of the merits of each paradigm.

$$
\begin{array}{rcl}
\langle \mathsf{Space} \rangle & ::= & \{NewSpace\ \ P\ \ X\} \\
& | & \{Choose\ \ N\ \ Y\} \\
& | & \{Ask\ \ X\ \ A\} \\
& | & \{Commit\ \ X\ \ I\} \\
& | & \{Clone\ \ X\ \ C\} \\
& | & \{Inject\ \ X\ \ P\} \\
& | & \{Merge\ \ X\ \ Y\}
\end{array}
$$

Figure 4: Primitive operations for computation spaces

Oz has syntactic support to make this style comfortable and implementation support to make it efficient. The same approach applies to more "formal" paradigms such as functional and logic programming. Strict functions are restricted versions of procedures in which the binding is directional. Non-strict (lazy) functions are implemented with `ByNeed`.

For logic programming, procedures become relations when they have a logical semantics in addition to their operational semantics. This is illustrated by many examples in this article, starting in Section 2. For example, the `if` and `case` statements are special cases of a logical operation, i.e., entailment checking, which is available in a more general form by the `cond` statement. To make the execution *complete*, i.e., to always find a constructive proof when one exists, it is necessary to add search. Oz supports search through computation spaces. When they are combined with the rest of the model, it is possible to efficiently express all the search one needs in Oz (see Section 7.4, below).

7.4 Computation spaces

Computation spaces are an abstraction that permits the high-level programming of search abstractions and deep guard combinators, both of which are important for constraint and logic programming. Spaces have the flexibility needed for real-world constraint problems and they can be implemented efficiently: on real-world problems the Mozart 1.1.0 implementation using copying and recomputation is competitive in time and memory use with traditional systems using trailing-based backtracking [66].

Spaces are a natural way to integrate search, which has a strong speculative component, into a concurrent system that interacts with its environment. We say an execution is *speculative* if it is not guaranteed to be part of the final computation. Managing speculative computation is the essence of nondeterministic logic programming. Spaces are compositional, i.e., they can be nested, which is an important property.

This section defines computation spaces, the operations that can be performed on them (see Figure 4), and gives examples of how to use them to program search. The discussion in this section follows the model in [68]. This model is implemented in Mozart 1.1.0 [54] and refines the one presented in the articles [65, 69]. The space abstraction can be made language-independent; [29] describes a C++ implementation of a similar abstraction that provides both trailing and copying.

7.4.1 Definition

A computation space is just an Oz store, with its three parts: thread store, constraint store, and mutable store. The store we have seen so far is a single computation space with equality constraints over rational trees. When dealing with search a more elaborate structure is used with multiple nested computation spaces over other constraint domains. Since spaces can be speculative, it is important to be precise about the visibility of variables and their bindings. Figure 5 gives an example. The general rules for the structure of computation spaces are as

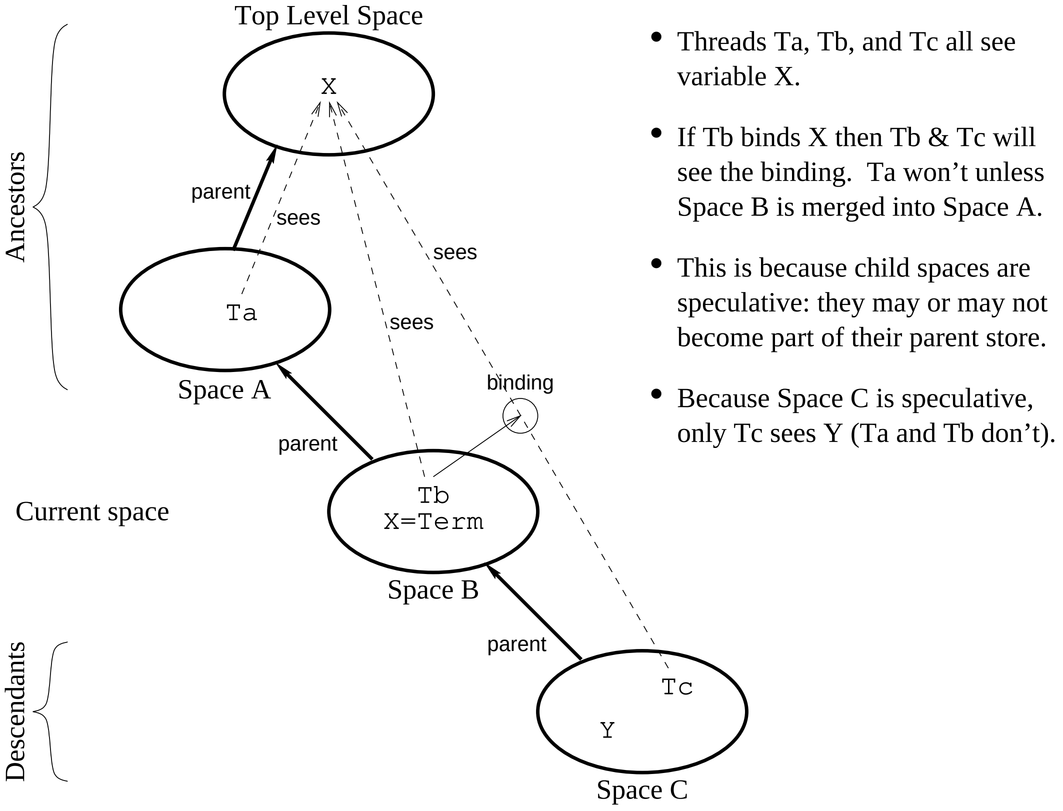

Figure 5: Visibility of variables and bindings in nested spaces

follows.

- There is always a *top level* computation space where threads may interact with the external world. The top level space is just the store of Section 7.1. Because the top level space interacts with the external world, its constraint store always remains consistent, that is, each variable has at most one binding that never changes once it is made. A thread that tries to add an inconsistent binding to the top level constraint store will raise a failure exception.

- A thread may create a new computation space. The new space is called a *child space*. The current space is the child's *parent space*. At any time, there is a tree of computation spaces in which the top level space is the root. With respect to a given space, a higher one in the tree (closer to the root) is called an *ancestor* and a lower one is called a *descendant*.

- A thread always belongs to exactly one computation space. A variable always belongs to exactly one computation space.

- A thread sees and may access variables belonging to its space as well as to all ancestor spaces. The thread cannot see the variables of descendant spaces.

- A thread cannot see the variables of a child space, unless the child space is *merged* with its parent. Space merging is an explicit program operation. It causes the child space to disappear and all the child's content to be added to the parent space.

- A thread may add bindings to variables visible to it. This means that it may bind variables belonging to its space or to its ancestor spaces. The binding will only be visible in the current space and its descendants. That is, the parent space does not see the binding unless the current space is merged with it.
- If a thread in a child space tries to add an inconsistent binding to its constraint store, then the space fails.

7.4.2 State of a space

A space is *runnable* if it or a descendant contains a runnable thread, and *blocked* otherwise. Let us run all these threads until the space is blocked. Then the space can be in one of the following further states:

- The space is *stable*. This means that no additional bindings done in an ancestor can make the space runnable. A stable space can be in four further states:
  - The space is *succeeded*. This means that it contains no choice points. A succeeded space contains a solution to the logic program.
  - The space is *distributable*. This means that the space has one thread that is suspended on a choice point with two or more alternatives. A space can have at most one choice point.
  - The space is *failed*. This is defined in the previous section; it means that the space attempted to bind the same variable to two different values. No further execution happens in the space.
  - The space is *merged*. This means that the space has been discarded and its constraint store has been added to a parent. Any further operation on the space is an error. This state is the end of a space's lifetime.
- The space is *not stable*. This means that additional bindings done in an ancestor can make the space runnable. Being not stable is usually a temporary condition due to concurrency. It means that some ancestor space has not yet transferred all required information to the space. A space that stays not stable indefinitely usually means a programmer error.

7.4.3 Programming search strategies

Search strategies are programmed with spaces in the following way:

- Create the space with the correct program inside.
- Let the program run inside the space. The program will do as much propagation as possible, but will eventually block when no more propagation can be done. At that point, it may possibly have created a choice point.
- Detect when the program can go no further, i.e., when the space has become stable. In that case, either stop and return with a solution, or continue execution by forcing the choice point to commit to an alternative.

The next section explains the operations we need for this approach. Then, Section 7.4.5 gives some examples of how to program search with them. Many other strategies can be programmed than are shown here; for more information see [68].

7.4.4 Space operations

Now we know enough to define the primitive space operations. There are seven principal ones (see Figure 4):

- `{NewSpace P X}`, when given a unary procedure `P`, creates a new computation space `X`. In this space, a fresh variable `R`, called the *root variable*, is created and a new thread, and `{P R}` is invoked in the thread.
- `{Choose N Y}` first creates in the current space a choice point with `N` alternatives. Then it blocks, waiting for an alternative to be chosen by a `Commit` operation on the space (see below). The `Choose` call defines only the *number* of alternatives; it does not specify what to do for any given alternative. `Choose` returns with `Y=I` when alternative `1`≤`I`≤`N` is chosen. A maximum of one choice point at a time may exist in a space.
- `{Ask X A}` asks the space `X` for its status. As soon as the space becomes stable, `A` is bound. If `X` is failed, merged, or succeeded, then `A` is bound to `failed`, `merged`, or `succeeded`. If `X` is distributable, then `A=alternatives(N)`, where `N` is the number of alternatives.
- `{Commit X I}`, if `X` is a distributable space, causes the `Choose` call in the space to complete and return `I` as its result. This may cause the space to resume execution. The integer `I` must satisfy `1`≤`I`≤`N`, where `N` is the first argument of the `Choose` call.
- `{Clone X C}`, if `X` is a stable space, creates an identical copy (a *clone*) of `X` in `C`. This allows both alternatives of a distributable space to be explored.
- `{Inject X P}` is similar to space creation except that it uses an existing space `X`. It creates a new thread in the space and invokes `{P R}` in the thread, where `R` is the space's root variable. This makes a stable space not stable again. Adding constraints to an existing space is necessary for some search strategies such as branch-and-bound and saturation.
- `{Merge X Y}` binds `Y` to the root variable of space `X` and discards the space.

7.4.5 Using spaces

These seven primitive operations are enough to define many search strategies and deep guard combinators [68]. Here are a few examples.

Depth-first search. Figure 6 shows how to program depth-first single solution search, in the case of binary choice points. This explores the search tree in depth-first manner and returns the first solution it finds. The problem is defined as a unary procedure `{Script Sol}` that gives a reference to the solution `Sol`, just like the example in Section 6.3. The solution is returned in a one-element list as `[Sol]`. If there is no solution, then `nil` is returned. In `Script`, choice points are defined with the `Choose` operation. A naive choice point, one that selects all its alternatives in order, can be defined as follows:

```
case {Choose N}
of 1 then S1
[] 2 then S2
...
[] N then Sn
end
```

This can be given the following syntax:

$$\texttt{choice}\ \langle \mathsf{S}_1 \rangle\ []\ \langle \mathsf{S}_2 \rangle\ []\ \ldots\ []\ \langle \mathsf{S}_n \rangle\ \texttt{end}$$

```
fun {DFE S}
   case {Ask S}
   of failed then nil
   [] succeeded then [S]
   [] alternatives(2) then C={Clone S} in
      {Commit S 1}
      case {DFE S} of nil then {Commit C 2} {DFE C}
      [] [T] then [T]
      end
   end
end

% Given procedure {Script Sol}, returns solution [Sol] or nil:
fun {DFS Script}
   case {DFE {NewSpace Script}} of nil then nil
   [] [S] then [{Merge S}]
   end
end
```

Figure 6: Depth-first single solution search

Many other search techniques and strategies can be programmed. Some basic techniques are n-ary search, all-solution search, and lazy search. Some basic strategies are limited discrepancy search, best first search, and branch-and-bound search. All these can be combined and parametrized in many ways.

Andorra-style disjunction (the `dis` statement). The previous example shows how to program the **`choice`** statement, which is a naive choice point. The **`dis`** statement is an optimization of **`choice`** that eliminates failed alternatives and commits immediately if there is a single remaining alternative (Section 3):

$$
\begin{array}{l}
\texttt{dis} \\
\quad Y_{1a} \;...\; Y_{1z} \text{ in } \langle \mathsf{G}_1 \rangle \text{ then } \langle \mathsf{S}_1 \rangle \\
\quad \texttt{[]} \;\; \texttt{...} \;\; \texttt{[]} \\
\quad Y_{na} \;...\; Y_{nz} \text{ in } \langle \mathsf{G}_n \rangle \text{ then } \langle \mathsf{S}_n \rangle \\
\texttt{end}
\end{array}
$$

The **`dis`** statement can be programmed with the space operations as follows. First encapsulate each guard of the **`dis`** statement in a separate space. Then execute each guard until it is stable. Discard all failed guards. Finally, using the `Choose` operation, create a choice point for the remaining guards. The exact definition of **`dis`** is a bit subtle; see [68] for details.

If **`dis`** is important for real programs, then its common cases can be made faster by using **`case`** to do preselection before doing any space operations. This will avoid using spaces in those cases where they are not really needed. In practice, we find that **`dis`** is almost never used outside of small examples, so this optimization is not worth the effort.

Deep guard combinators. With spaces, it is possible to program constraint combinators that are both compositional and conservative. This is a powerful way to combine constraints from different domains and to prototype new constraints. Some examples of combinators are deep negation, constraint reification, propagation-based disjunction (such as **`dis`**), and deep committed-choice (deep **`cond`**).

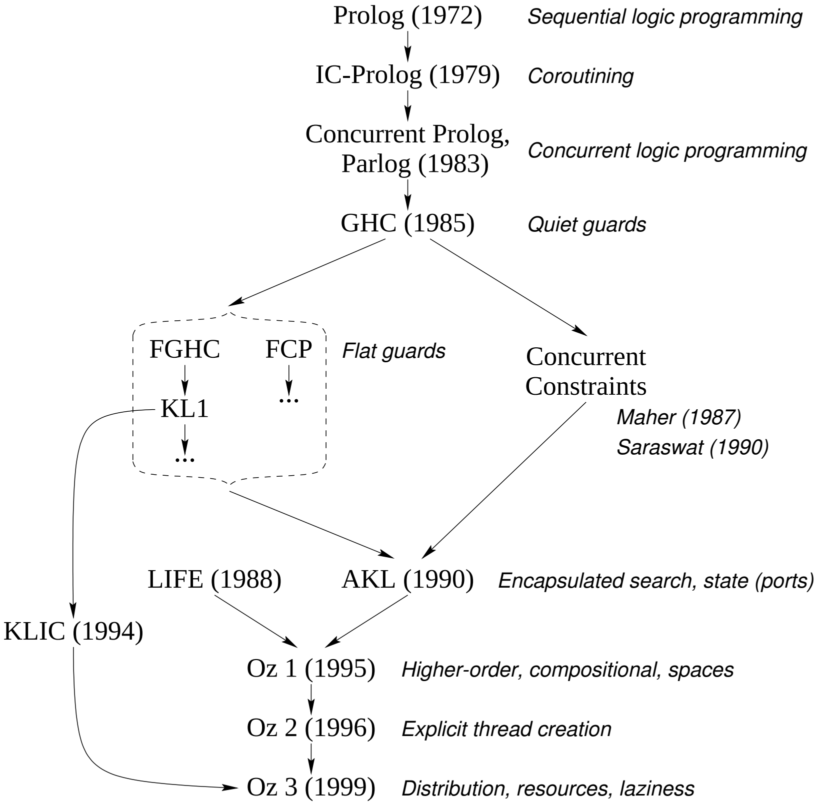

Figure 7: History sketch of Oz

8 History sketch

Oz is the latest in a long line of logic-based languages that originated with Prolog (see Figure 7). We summarize briefly the evolutionary path and give some of the important milestones along the way. First experiments with concurrency were done in the venerable IC-Prolog language where coroutining was used to simulate concurrent processes [12, 13]. This led to Parlog and Concurrent Prolog, which introduced the process model of logic programming, usually known as *concurrent* logic programming [11, 71, 72]. The advent of GHC (Guarded Horn Clauses) simplified concurrent logic programming considerably by introducing the notion of *quiet guards* [84]. A clause matching a goal will fire only if the guard is entailed by the constraint store. This formulation and its theoretical underpinning were pioneered by the work of Maher and Saraswat as they gave a solid foundation to concurrent logic programming [44, 61, 62]. On the practical side, systems with "flat" guards (which are limited to basic constraints or system-provided tests) were the focus of much work [83]. The flat versions of Concurrent Prolog and GHC, called FCP and FGHC respectively, were developed into large systems [34, 73]. The KL1 (Kernel Language 1) language, derived from FGHC, was implemented in the high-performance KLIC system. This system runs on sequential, parallel, and distributed machines [21]. Some of the implementation techniques in the current Mozart system have been borrowed from KLIC, notably the distributed garbage collection algorithm.

An important subsequent development was AKL (Andorra Kernel Language) [36, 35, 37], which added explicit state (in the form of ports), encapsulated search, and an efficient implementation of deep guards. AKL provides the first synthesis of concurrent and constraint logic programming. Search is encapsulated by using nested computation spaces. A computation space is a constraint store with its associated goals. Search is done by allowing procedures to be defined by a sequence of don't-know guarded clauses. These definitions denote disjunctions. When local propagation cannot choose between different disjuncts, then the program can try each disjunct by cloning the computation space.

The initial Oz language, Oz 1, was largely derived from AKL and LIFE, but added the notions of higher-order procedures, more controllable search by making computation spaces first class, compositional syntax, and the cell primitive for mutable state [74]. LIFE integrates logic and functional programming based on the concept of *residuation*, a delaying mechanism very similar to entailment-driven guards [4, 5, 3, 2]. Oz 1 takes the record data type from LIFE [92]. Concurrency in Oz 1 is implicit and based on lazy thread creation. When a statement suspends, a new thread is created that contains only the suspended statement. The main thread is not suspended but continues with the next statement.

The direct successor of Oz 1, namely Oz 2, abandons implicit concurrency in favor of an explicit thread creation construct. Thread suspension and resumption are still based on dataflow using logic variables. Our experience shows that explicit concurrency makes it easier for the user to control application resources. It allows the language to have an efficient and expressive object-oriented model without sequential state threading within method definitions. It allows a simple debugging model and it makes it easy to add exception handling to the language.

The current Oz language, Oz 3, conservatively extends Oz 2 with support for first-class module specifications, called *functors* [20], and for open, robust, distributed programming [26, 90, 25, 88, 77]. A functor specifies a module in terms of the other modules it needs. Distribution is transparent, i.e., the language semantics is unchanged independent of how the program is distributed. With respect to logic programming, the distributed extension has two properties:

- The top level space is efficiently distributed over multiple processes. In particular, the top level store is implemented by a practical algorithm for distributed rational tree unification [25].
- A child computation space is a stationary entity that exists completely in one process. Child spaces are extremely lightweight. Due to the communication overheads involved, we have not found it worthwhile to distribute one child space over multiple processes. Constraint propagation *within* a child space is therefore completely centralized. Parallel search engines (see example in Section 6.3) are implemented by putting child spaces in different processes.

In all versions of Oz, concurrency is intended primarily to model logical concurrency in the application rather than to achieve parallelism (speedup) in the implementation. However, the distributed implementation is still useful for parallel execution. It is optimized to be particularly efficient on shared-memory multiprocessors. In that case, interprocess communication is implemented using shared pages between address spaces [26].

9 Lessons learned

One of the goals of the Oz project was to use logic programming for real-world problems. During the course of the project, we have tried out many implementations and programming techniques, and built many applications. From this long experience, we have learned many hard lessons both for practical logic programming and for multiparadigm programming. Here is a summary of the most important of these lessons. We agree with the conclusions of Hughes, namely that higher-order procedures are essential and that laziness is extremely useful [33].

9.1 Be explicit ("magic" does not work)

- Provide explicit concurrency (other concurrent logic programming systems have implicit concurrency). This is important for interaction with the environment, efficiency, facilitating reasoning (e.g., for termination), and debugging. It is also important for distributed programming.

- Provide explicit search (Prolog has implicit search). To put it bluntly, 99% of Prolog programs solve algorithmic problems, which don't need search, yet one can't use Prolog without learning about search. Furthermore, for search problems the search must be *very* controllable, otherwise it does not scale to real applications. Prolog's implicit search is much too weak; this means that inefficient approaches such as meta-interpreters are needed. We conclude that Prolog's search is ineffective for both algorithmic and search problems, and therefore it has no justification at all.

- Provide explicit state (in C++ and Java, state is implicit, e.g., Java variables are stateful unless declared `final`). Explicit state is crucial for programs that interact with their environments. Furthermore, it is important for reasoning (e.g., liveness properties) and for distributed programming.

- Provide explicit laziness (in Haskell, laziness is implicit for all functions). Explicitly declaring functions as lazy makes them easy to implement and documents the programmer's intention.

9.2 Provide primitives for building abstractions

- Full compositionality is essential: everything can be nested everywhere. This implies higher-order procedures with lexical scoping. Carefully design new abstractions to be fully compositional.

- The language should make it easy to define new abstractions. The developer has all the primitives necessary to build powerful abstractions of his own. For example, in addition to lexical scoping, it is important to have *read-only* logic variables, which allow to build abstractions that export logic variables and still protect them [50]. There is no distinction between built-in abstractions and application-specific ones, except possibly regarding performance. Examples of built-in abstractions are the object system, reentrant locks, distribution support, and user interface support.

9.3 Factorize and be lean

Complexity is a source of problems and must be reduced as much as possible:

- Factorize the design at *all* levels of abstraction, both in the language and the implementation. Keep the number of primitive operations to a minimum.

- It is important to have a sophisticated module system, with lazy loading and support for mutually-dependent modules. In Mozart, both Oz and C++ modules can be loaded lazily, i.e., only when the module is needed. In this way, the system is both lean and has lots of functionality. Lazy loading of Oz modules is implemented with the `ByNeed` operation (see Section 7.2). Support for mutually-dependent Oz modules means that cyclic dependencies need to bottom out only at run-time, not at load-time. This turns out to be important in practice.

- It is important to have a powerful interface to a lower-level language. Mozart has a C++ interface that allows to add new constraint systems [49, 56]. These constraint systems are fully integrated into the system, including taking advantage of the full power of computation spaces. Mozart also supports garbage collection across the interface, both from the Oz side (using finalization and weak pointers) and the C++ side.

9.4 Support true multiparadigm programming

In any large programming project, it is almost always a good idea to use more than one paradigm:

- Different parts are best programmed in different paradigms[5]. For example, an event handler may be defined as an active object whose new state is a function of its previous state and an external event. This uses both the object-oriented and functional paradigms.

- Different levels of abstraction require different paradigms. For example, consider a multi-agent system programmed in a concurrent logic language. At the language level, the system may have a logical semantics. But there is also a higher level, the agent level, consisting of stateful entities called "agents" sending messages to each other. Strictly speaking, these concepts do not exist at the logical level. To reason about them, the agent level is better specified as a graph of active objects.

It is always possible to *encode* one paradigm in terms of another. In the case of pure concurrent logic programs and *state*, we explain why this is not a good idea [37]. The canonical way to encode state in a pure concurrent logic program is by using *streams*. An active object is a recursive predicate that reads an internal stream. The object's current state is this stream's most-recently-calculated element. A *reference* to an active object is a stream that is read by that object. This reference can only be used by one sender object, which sends messages by binding the stream's tail. Two sender objects sending messages to a third object are coded as two streams feeding a *stream merger*, whose output stream then feeds the third object. Whenever a new reference is created, a new stream merger has to be created. The system as a whole is therefore more complex:

- The communications graph of the active objects is encoded as a network of streams and stream mergers. In this network, each object has a tree of stream mergers feeding into it. The trees are created incrementally during execution, as object references are passed around the system.

- To regain efficiency, the compiler and run-time system must be smart enough to discover that this network is equivalent to a much simpler structure in which senders send directly to receivers. This "decompilation" algorithm is so complex that to our knowledge no pure concurrent logic system implements it.

On the other hand, adding state directly to the execution model makes the system simpler and more uniform. In that case, programmer-visible state (e.g., active objects) is mapped directly to execution model state (e.g., threads and cells), which is compiled directly into machine state. Both the compiler and the run-time system are simple. The counterargument that the stateful execution model is no longer "pure" is true but of no practical use, since the stateful model allows simpler reasoning than the "pure" stateless one.

Similar examples can be found for other concepts, e.g., higher-orderness, concurrency, exception handling, search, and laziness. In each case, encoding the concept increases the complexity of both the program and the system implementation. In each case, adding the concept to the execution model gives a simpler and more uniform system. We conclude that the project's programming language should support multiple paradigms.

[5]In this case, another approach is to use multiple languages with well-defined interfaces. This is more complex, but can sometimes work well.

9.5 Combine dynamic and static typing

Oz is a strongly-typed language that is dynamically typed, i.e., it has an *enforced* type structure that is checked at *run-time*. In a statically-typed language, on the other hand, the type structure is checked at compile-time. Dynamic typing puts many fewer restrictions on programs and programming than static typing. For example, it allows Oz to have an incremental development environment that is part of the run-time system. It allows truly open programming, i.e., independently-written components can come together and interact with as few assumptions as possible about each other. It makes reflective programming easy. By *reflective programming* we mean the ability to expose part of the implementation to the application programmer, such as in meta-object protocols [39] and first-class constraints [57].

Static typing is usually presented as having two advantages with respect to dynamic typing: it allows to catch more program errors and it allows a more efficient implementation. We find that these advantages are less important than they appear. First of all, static and dynamic typing are roughly equivalent in practical error-catching ability. While static typing allows earlier detection of some classes of program errors than dynamic typing, this is offset by two factors: first, the incremental development environment permitted by dynamic typing and second, not having to reformulate the program to fit within the limitations of the type system. In our experience, the *really tough* errors, i.e., the ones that take up most debugging time, are not type errors at all, but come from deeper misunderstandings about semantics.

Static typing permits a more efficient implementation than dynamic typing, since the compiler has more information with which it can choose a representation. However, this increased efficiency is only needed in a very small part of the program. For this reason, we find that the decreased flexibility of static typing, which is felt in the *whole* program, is not offset by this increased efficiency. Efficiency is an issue only in critical parts of a program, which are partly known during the design phase, but which must be confirmed by profiling real executions. Only these parts need be rewritten for speed.[6]

In conclusion, we do not yet have a satisfactory solution that combines the advantages of static and dynamic typing. It seems that the right type system should be "mixed". It would allow dynamic typing by default, complemented by optional static typing in small, well-defined parts of the program. Much work has been done to add some of the advantages of dynamic typing to a statically-typed language, while keeping the good properties of static typing:[7]

- Polymorphism adds flexibility to functional and object-oriented languages.
- Type inferencing, pioneered by ML, relieves the programmer of the burden of having to type the whole program explicitly.

In a mixed type system, on the other hand, the default would be dynamic typing, not static typing. The static typing is only put where it is needed. For example, module interfaces and procedure arguments could be statically typed to maximize early detection of errors. The most-executed part of a program could be statically typed to maximize its efficiency. In a mixed type system, the trade-off between flexibility and efficiency is not frozen by the language design, but is available to the programmer. The design of this mixed type system is a subject for future research.

9.6 Use an evolutionary development methodology

The development methodology used in the Oz project has been refined over many years, and is largely responsible for the combination of expressive power, semantic simplicity, and

[6]This is summarized in the folk "Rules of Optimization". The Zeroth Rule of Optimization states: The greatest performance increment comes in the step from not working to working (attributed to John Ousterhout). The First Rule of Optimization states: Don't. The Second Rule of Optimization states: Not Yet.

[7]Type casts like in C++ are another approach, but we reject them since they break all program invariants.

implementation efficiency in Mozart. The methodology is nowhere fully described in print; there are only partial explanations [74, 88]. We summarize it here.

The main idea is that the system's *design* is never finished, while there is (in principle) always a working *implementation*. The system's developers continuously introduce new abstractions as solutions to practical problems. The burden of proof is on the developer proposing the abstraction: he must prototype it and show an application for which it is necessary. The net effect of a new abstraction must be either to simplify the system or to greatly increase its expressive power. If this seems to be the case, then intense discussion takes place among all developers to simplify the abstraction as much as possible. Often it vanishes: it can be completely expressed without modifying the system. This is not always possible. Sometimes it is better to modify the system: to extend it or to replace an existing abstraction by a new one.

The decision whether to accept an abstraction is made according to several criteria including aesthetic ones. Two major acceptance criteria are related to implementation and formalization. The abstraction is acceptable only if its implementation is efficient and its formalization is simple.

This methodology extends the approaches put forward by Hoare, Ritchie, and Thompson [30, 59, 81]. Hoare advocates designing a program and its specification concurrently. He also explains the importance of having a simple core language. Ritchie advises having the designers and others actually use the system during the development period. In Mozart, as in most Prolog systems, this is possible because the development environment is part of the run-time system. Thompson shows the power of a well-designed abstraction. The success of Unix was made possible due to its simple, powerful, and appropriate abstractions.

With respect to "traditional" software design processes, this methodology is closest to *exploratory programming* [78]. The main defect of exploratory programming, that it results in systems with ill-defined structure, is avoided by the way the abstractions are refined and by the requirement of simple formalization.

The two-step process of first generating abstractions and then selecting among them is analogous to the basic process of evolution. In evolution, an unending source of different individuals is followed by a filter, survival of the fittest [16]. In the analogy, the individuals are abstractions and the filters are the acceptance criteria of implementation and formalization. Some abstractions thrive (e.g., compositionality with lexical scoping), others die (e.g., the "generate and test" approach to search is dead, being replaced by propagate and distribute), others are born and mature (e.g., dynamic scope, which is currently under discussion), and others become instances of more general ones (e.g., deep guards, once basic, are now implemented with spaces).

10 Conclusions and perspectives

The Oz language provides powerful tools for both the algorithmic and search classes of logic programming problems. In particular, there are many tools for taming search in real-world situations. These tools include global constraints, search heuristics, and interactive libraries to visualize and guide the search process.

Oz is based on a lean execution model that subsumes deterministic logic programming, concurrent logic programming, nondeterministic logic programming, constraint programming, strict and lazy functional programming, and concurrent object-oriented programming. Because of appropriate syntactic and implementation support, all these paradigms are easy to use. We say that Oz is *multiparadigm*. To a competent Oz programmer, the conventional boundaries between these paradigms are artificial and irrelevant. It is important to be multiparadigm because good program design often requires different paradigms to be used for different parts of a program.

The Mozart system implements Oz and is in continuing development by the Mozart Consortium [54]. Research and development started in 1991. In addition to ongoing research in constraint programming, we are doing research in distribution, fault tolerance, security, transactions, persistence, programming environments, software component architectures, tools for collaborative applications, and graphic user interface programming. Another important topic, as yet unexplored, is the design of a mixed type system that combines the advantages of static and dynamic typing. The work on distribution and related areas started in 1995 [77]. Most of these areas are traditionally given short shrift by the logic and functional programming communities, yet they merit special attention due to their importance for real-world applications.

Acknowledgements

This article is based on the work of many people over many years. We thank Juris Reinfelds and all the contributors and developers of the Mozart system. Many of the opinions expressed are shared by other members of the Mozart Consortium. We thank Danny DeSchreye for suggesting the ICLP99 tutorial on which this article is based. We thank Krzysztof Apt, Manuel Hermenegildo, Kasunori Ueda, and others for their questions and comments at the ICLP99 tutorial where the original talk was given. This research is partly financed by the Walloon Region of Belgium.

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Mass, 1985.

[2] Hassan Aït-Kaci, Bruno Dumant, Richard Meyer, Andreas Podelski, and Peter Van Roy. The Wild LIFE handbook, 1994. Available at `http://www.info.ucl.ac.be/~pvr/`.

[3] Hassan Aït-Kaci and Patrick Lincoln. LIFE: A natural language for natural language. MCC Technical Report ACA-ST-074-88, MCC, ACA Program, February 1988.

[4] Hassan Aït-Kaci and Roger Nasr. Integrating logic and functional programming. *Journal of Lisp and Symbolic Computation*, 2:51–89, 1989.

[5] Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16(3-4):195–234, July-August 1993.

[6] J. Armstrong, M. Williams, C. Wikström, and R. Virding. *Concurrent Programming in Erlang.* Prentice-Hall, Englewood Cliffs, N.J., 1996.

[7] Robert L. Ashenhurst and Susan Graham, editors. *ACM Turing Award Lectures: The First Twenty Years.* ACM Press, 1987.

[8] Mats Carlsson *et al.* SICStus Prolog 3.8.1, December 1999. Available at `http://www.sics.se`.

[9] Yves Caseau, François-Xavier Josset, and François Laburthe. CLAIRE: Combining sets, search and rules to better express algorithms. In *International Conference on Logic Programming (ICLP 99)*, pages 245–259. MIT Press, November 1999.

[10] Yves Caseau, François Laburthe, and Glenn Silverstein. A meta-heuristic factory for vehicle routing problems. In Joxan Jaffar, editor, *Constraint Programming (CP 99)*, pages 144–158, Alexandria, VA, USA, 1999.

[11] Keith L. Clark. PARLOG: the language and its applications. In A. J. Nijman J. W. de Bakker and P. C. Treleaven, editors, *Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE). Volume II: Parallel Languages*, volume 259 of *LNCS*, pages 30–53, Eindhoven, The Netherlands, June 1987. Springer.

[12] Keith L. Clark and Frank McCabe. The control facilities of IC-Prolog. In D. Michie, editor, *Expert Systems in the Micro-Electronic Age*, pages 122–149. Edinburgh University Press, Edinburgh, Scotland, 1979.

[13] Keith L. Clark, Frank G. McCabe, and Steve Gregory. IC-PROLOG — language features. In Keith L. Clark and Sten-Åke Tärnlund, editors, *Logic Programming*, pages 253–266. Academic Press, London, 1982.

[14] Alain Colmerauer. PROLOG II reference manual and theoretical model. Technical report, Groupe d'Intelligence Artificielle, Université Aix - Marseille II, October 1982.

[15] Guy Cousineau and Michel Mauny. *The Functional Approach to Programming*. Cambridge University Press, December 1998.

[16] Charles Darwin. *On the Origin of Species by means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. Harvard University Press (originally John Murray, London), 1859.

[17] Denys Duchier. Axiomatizing dependency parsing using set constraints. In *Sixth Meeting on Mathematics of Language*, Orlando, Florida, July 1999.

[18] Denys Duchier, Claire Gardent, and Joachim Niehren. Concurrent constraint programming in Oz for natural language processing. Lecture notes, `http://www.ps.uni-sb.de/~niehren/oz-natural-language-script.html`, 1999.

[19] Denys Duchier, Leif Kornstaedt, Tobias Müller, Christian Schulte, and Peter Van Roy. System modules. Technical report, January 1999. Available at `http://www.mozart-oz.org`.

[20] Denys Duchier, Leif Kornstaedt, Christian Schulte, and Gert Smolka. A Higher-order Module Discipline with Separate Compilation, Dynamic Linking, and Pickling. Technical report, Programming Systems Lab, DFKI and Universität des Saarlandes, 1998. DRAFT.

[21] Tetsuro Fujise, Takashi Chikayama, Kazuaki Rokusawa, and Akihiko Nakase. KLIC: A portable implementation of KL1. In *Fifth Generation Computing Systems (FGCS '94)*, pages 66–79, December 1994.

[22] Michael R. Garey and David S. Johnson. *Computers and Intractibility*. W.H. Freeman and Co., 1979.

[23] Seif Haridi and Per Brand. Andorra Prolog – an integration of Prolog and committed-choice languages. In Institute of New Generation Computer Technology (ICOT), editor, the *International Conference on Fifth Generation Computer Systems (FGCS)*, volume 2, pages 745–754, November 1988.

[24] Seif Haridi and Sverker Janson. Kernel Andorra Prolog and its computation model. In *7th International Conference on Logic Programming*, pages 31–48. MIT Press, June 1990.

[25] 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*, 21(3):569–626, May 1999.

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

[27] Martin Henz. *Objects for Concurrent Constraint Programming*, volume 426 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston, November 1997.

[28] Martin Henz. *Objects in Oz*. PhD thesis, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany, June 1997.

[29] Martin Henz, Tobias Müller, and Ka Boon Ng. Figaro: Yet another constraint programming library. In *Workshop on Parallelism and Implementation Technology for Constraint Logic Programming, International Conference on Logic Programming (ICLP 99)*, 1999.

[30] Charles Antony Richard Hoare. The emperor's old clothes. In Ashenhurst and Graham [7]. 1980 Turing Award Lecture.

[31] Bruce Holmer, Bart Sano, Michael Carlton, Peter Van Roy, Ralph Haygood, William Bush, Alvin Despain, Joan Pendleton, and T.P. Dobry. Fast Prolog with an extended general purpose architecture. In *17th International Symposium on Computer Architecture (ISCA)*, pages 282–291. IEEE Computer Society Press, May 1990.

[32] Paul Hudak, Simon L. Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairnbairn, Joseph H. Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Richard B. Kieburtz, Rishiyur S. Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell, a non-strict purely functional language. *ACM SIGPLAN Notices*, 27(5):R1–R164, 1992.

[33] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[34] Institute for New Generation Computer Technology, editor. *Fifth Generation Computer Systems 1992*, volume 1,2. Ohmsha Ltd. and IOS Press, 1992. ISBN 4-274-07724-1.

[35] Sverker Janson. *AKL–A Multiparadigm Programming Language*. PhD thesis, Uppsala University and SICS, 1994.

[36] Sverker Janson and Seif Haridi. Programming paradigms of the Andorra Kernel Language. In *International Symposium on Logic Programming*, pages 167–183, October 1991.

[37] Sverker Janson, Johan Montelius, and Seif Haridi. Ports for objects in concurrent logic programs. In *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.

[38] Andreas Kågedal, Peter Van Roy, and Bruno Dumant. Logical State Threads 0.1, January 1997. Available at `http://www.info.ucl.ac.be/people/PVR/implementation.html`.

[39] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[40] Alexander Koller and Joachim Niehren. Constraint programming technology in computational linguistics. Technical report, Universität des Saarlandes, Programming Systems Lab, 2000. Submitted. Available at `http://www.ps.uni-sb.de/Papers/abstracts/CP-NL.html`.

[41] François Laburthe and Yves Caseau. SaLSA: A language for search algorithms. In *Constraint Programming (CP 98)*, Lecture Notes in Computer Science, vol. 1520, pages 310–324. Springer Verlag, 1998.

[42] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. In *Second International Symposium on Operating Systems, IRIA*, October 1978. Reprinted in *Operating Systems Review*, 13(2), April 1979, pp. 3–19.

[43] Benjamin Lorenz. The Mozart debugger. Technical report, January 1999. Available at `http://www.mozart-oz.org`.

[44] Michael Maher. Logic semantics for a class of committed-choice programs. In *International Conference on Logic Programming (ICLP 87)*, pages 858–876, May 1987.

[45] David Maier and David S. Warren. *Computing with logic: Logic Programming with Prolog*. Addison-Wesley, January 1988.

[46] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, January 1999.

[47] Martin Müller, Tobias Müller, and Peter Van Roy. Multiparadigm programming in Oz. In Donald Smith, Olivier Ridoux, and Peter Van Roy, editors, *Workshop on the Future of Logic Programming, International Logic Programming Symposium (ILPS 95)*, December 1995.

[48] Michael Mehl. *The Oz Virtual Machine - Records, Transients, and Deep Guards*. PhD thesis, Technische Fakultät der Universität des Saarlandes, 1999.

[49] Michael Mehl, Tobias Müller, Christian Schulte, and Ralf Scheidhauer. Interfacing to C and C++. Technical report, January 2000. Available at `http://www.mozart-oz.org`.

[50] Michael Mehl, Christian Schulte, and Gert Smolka. Futures and by-need synchronization for Oz. May 1998.

[51] Bertrand Meyer. *Object-oriented Software Construction*. Prentice-Hall, 1988.

[52] Robin Milner, Mads Tofte, and Robert Harper. *Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[53] Chris Moss. *Prolog++: The Power of Object-Oriented and Logic Programming*. Addison-Wesley, 1994.

[54] Mozart Consortium. The Mozart programming system (Oz 3), version 1.1.0, January 2000. Available at `http://www.mozart-oz.org`.

[55] Tobias Müller. Problem solving with finite set constraints in Oz. A tutorial. Technical report, January 1999. Available at `http://www.mozart-oz.org`.

[56] Tobias Müller. The Mozart constraint extensions tutorial. Technical report, January 2000. Available at `http://www.mozart-oz.org`.

[57] Tobias Müller. Promoting constraints to first-class status. In *Proceedings of the First International Conference on Computational Logic*, London, 2000. To appear.

[58] Konstantin Popov. The Oz Browser. Technical report, January 1999. Available at `http://www.mozart-oz.org`.

[59] Dennis M. Ritchie. Reflections on software research. In Ashenhurst and Graham [7]. 1983 Turing Award Lecture.

[60] Vitor Santos Costa, David H. D. Warren, and Rong Yang. Andorra-I: A parallel Prolog system that transparently exploits both And- and Or-parallelism. In 3rd ACM SIGPLAN Conference on Principles and Practice of Parallel Programming, pages 83–93, August 1991.

[61] Vijay Saraswat and Martin Rinard. Concurrent constraint programming. In *POPL 90*, pages 232–245, January 1990.

[62] Vijay A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

[63] Richard D. Schlichting and Vicraj T. Thomas. A multi-paradigm programming language for constructing fault-tolerant, distributed systems. Technical Report TR 91-24, Dept. of Computer Science, University of Arizona, October 1991.

[64] Christian Schulte. Oz Explorer: A visual constraint programming tool. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, July 1997. MIT Press.

[65] 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.

[66] Christian Schulte. Comparing trailing and copying for constraint programming. In *International Conference on Logic Programming (ICLP 99)*, pages 275–289. MIT Press, November 1999.

[67] Christian Schulte. Oz Explorer–Visual constraint programming support. Technical report, January 1999. Available at `http://www.mozart-oz.org`.

[68] Christian Schulte. *Programming Constraint Inference Services*. PhD thesis, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany, 2000.

[69] Christian Schulte. Programming deep concurrent constraint combinators. In Enrico Pontelli and Vtor Santos Costa, editors, *Practical Aspects of Declarative Languages, Second International Workshop, PADL 2000*, volume 1753 of *Lecture Notes in Computer Science*, pages 215–229, Boston, MA, USA, January 2000. Springer-Verlag.

[70] Christian Schulte and Gert Smolka. Finite domain constraint programming in Oz. A tutorial. Technical report, January 1999. Available at `http://www.mozart-oz.org`.

[71] Ehud Shapiro. A subset of Concurrent Prolog and its interpreter. Technical Report TR-003, Institute for New Generation Computer Technology (ICOT), Cambridge, Mass., January 1983.

[72] Ehud Shapiro, editor. *Concurrent Prolog: Collected Papers*, volume 1-2. MIT Press, Cambridge, Mass., 1987.

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

[74] Gert Smolka. The Oz programming model. In *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer Verlag, Berlin, 1995.

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

[76] Gert Smolka, Martin Henz, and Jörg Würtz. Object-oriented concurrent constraint programming in Oz. In Pascal Van Hentenryck and Vijay Saraswat, editors, *Principles and Practice of Constraint Programming*, pages 29–48, Cambridge, Mass., 1995. MIT Press.

[77] Gert Smolka, Christian Schulte, and Peter Van Roy. PERDIO—Persistent and distributed programming in Oz. BMBF project proposal. Available at `http://www.ps.uni-sb.de`, February 1995.

[78] Ian Sommerville. *Software Engineering*. Addison-Wesley, 1992.

[79] Guy L. Steele, Jr. *Common LISP: The Language*. Digital Press, 1984.

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

[81] Ken Thompson. Reflections on trusting trust. In Ashenhurst and Graham [7]. 1983 Turing Award Lecture.

[82] Simon Thompson. *Haskell: The Craft of Functional Programming, Second Edition*. Addison-Wesley, June 1999.

[83] Evan Tick. The deevolution of concurrent logic programming. *J. Log. Prog.*, 23(2):89–123, May 1995.

[84] Kasunori Ueda. Guarded Horn Clauses. In Eiti Wada, editor, *Proceedings of the 4th Conference on Logic Programming*, volume 221 of *LNCS*, pages 168–179, Tokyo, Japan, July 1985. Springer.

[85] Pascal Van Hentenryck. *The OPL Programming Language*. MIT Press, February 1999. Software available from ILOG France.

[86] Peter Van Roy. A useful extension to Prolog's Definite Clause Grammar notation. *ACM SIGPLAN Notices*, 24(11):132–134, November 1989.

[87] Peter Van Roy. Logic programming in Oz with Mozart. In *International Conference on Logic Programming (ICLP 99)*, pages 38–51. MIT Press, November 1999. Mild version.

[88] Peter Van Roy. On the separation of concerns 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.

[89] Peter Van Roy and Alvin Despain. High-performance logic programming with the Aquarius Prolog compiler. *IEEE Computer*, pages 54–68, January 1992.

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

[91] 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.

[92] Peter Van Roy, Michael Mehl, and Ralf Scheidhauer. Integrating efficient records into concurrent constraint programming. In *International Symposium on Programming Languages, Implementations, Logics, and Programs*, Aachen, Germany, September 1996. Springer-Verlag.

[93] Philip Wadler. The essence of functional programming. In *POPL 92*, pages 1–14, 1992.

Contents

1 Introduction **1**

2 Deterministic logic programming **2**

3 Nondeterministic logic programming **4**

4 Concurrent logic programming **5**
4.1 Implicit versus explicit concurrency 6
4.2 Concurrent producer-consumer 6
4.3 Lazy producer-consumer 6
4.4 Coroutining 7

5 State **7**
5.1 Explicit state creation 8
5.2 Creating an active object 8

6 More on search **9**
6.1 Aggregate search 9
6.2 Simple search procedures 10
6.3 The scalable way to do search 11

7 The Oz execution model **12**
7.1 The store 13
7.2 The kernel language 14
7.3 Multiparadigm programming 15
7.4 Computation spaces 17
7.4.1 Definition 17
7.4.2 State of a space 19
7.4.3 Programming search strategies 19
7.4.4 Space operations 20
7.4.5 Using spaces 20

8 History sketch **22**

9 Lessons learned **23**
9.1 Be explicit ("magic" does not work) 24
9.2 Provide primitives for building abstractions 24
9.3 Factorize and be lean 24
9.4 Support true multiparadigm programming 25
9.5 Combine dynamic and static typing 26
9.6 Use an evolutionary development methodology 26

10 Conclusions and perspectives **27**