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

PETER VAN ROY
Université catholique de Louvain, B-1348 Louvain-la-Neuve, Belgium

PER BRAND
Swedish Institute of Computer Science, S-164 28 Kista, Sweden

DENYS DUCHIER
Universität des Saarlandes, D-66123 Saarbrücken, Germany

SEIF HARIDI
Royal Institute of Technology (KTH), S-164 28 Kista, Sweden

MARTIN HENZ
National University of Singapore, Singapore 117543

CHRISTIAN SCHULTE
Royal Institute of Technology (KTH), S-164 28 Kista, Sweden

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 formulate 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 problem 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 less than optimal 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

∗ 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.

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. 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 totally different approach:

- **Algorithmic problems**. These are problems for which efficient algorithms are known. This includes 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* may be natural ways to express it. Logical specifications are not always simpler. Sometimes an imperative specification is better, 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 or because such algorithms have not been made explicit. We cite, e.g., NP-complete problems (Garey and Johnson 1979) or problems with complex specifications whose algorithms are difficult for this reason. 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. This is because the size of the search space grows exponentially with the problem size. For a real application, all possible effort must be made to reduce the need for search: use strong (global) constraints, concurrency for cooperative constraints, heuristics for the search tree, etc. (Schulte and Smolka 1999). For problems with complex specifications, using sufficiently strong constraints sometimes results in a polynomial-time algorithm (Koller and Niehren 2000).

For this paper, we consider *logic programming* as programming with executable specifications written in a simple logic such as first-order predicate calculus. The Oz support for logic programming is targeted specifically towards the two classes of algorithmic problems 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 state. In our experience, 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 concurrency, lazy evaluation, exception handling, security, state, and search. 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 an overview of other research in multiparadigm programming and a short history of Oz. Finally, Section 9 summarizes the lessons we have learned in the Oz project on how to do practical logic programming and multiparadigm programming.

This article gives an informal (yet precise) introduction targeted towards Prolog programmers. A more complete presentation of logic programming in Oz and its relationship to other programming concepts is given in the textbook (Van Roy and Haridi 2002).

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:

```
declare
proc {Append Xs Ys Zs}
   case Xs
   of nil then Zs=Ys
   [] 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 (Sterling and Shapiro 1986; Maier and Warren 1988;

Thompson 1999; Cousineau and Mauny 1998). We explain it briefly, pointing out where it differs from Prolog. All capitalized identifiers refer to logic variables in a constraint store.[1] `Append` and `NRev` are procedures whose arguments are passed by unification, as in Prolog. The **`declare`** declares new global identifiers, `Append` and `NRev`, which are bound to the newly-created procedure values. This means that the order of the declarations does not matter. All local variables must be declared within a scope, e.g., "`Zr` **`in`**" and "`R` **`in`**" declare `Zr` and `R` with scopes to the next enclosing **`end`** keyword. A list is either the atom `nil` or a pair of an element `X` and a list `Xr`, written `X|Xr`. The `[]` is not an empty list, but separates clauses in a **`case`** statement (similar to a guarded command, except that **`case`** is sequential).

We explain briefly the semantics of the naive reverse to highlight the relationship to logic programming. The constraint store consists of equality constraints over rational trees, similar to what is provided by many modern Prolog systems. Statements are executed sequentially. There are two basic operations on the store, ask and tell (Saraswat 1993):

- The tell operation (e.g., `Ys=nil`) adds a constraint; it performs unification. The tell is an *incremental tell*; if the constraint is inconsistent with the store then only a consistent part is added and an exception is raised (see, e.g., (Smolka 1995b) for a formal definition).
- The ask operation is the **`case`** statement (e.g., **`case`** `Xs` **`of`** `X|Xr` **`then ... else ... end`**). It waits until the store contains enough information to decide whether the pattern is matched (entailment) or can never be matched (disentailment).

The above example can be written in a functional syntax. We find that a functional syntax often greatly improves the readability of programs. It is especially useful when it follows the data flow, i.e., the input and output arguments. In Oz, the definition of `NRev` in functional syntax is as follows:

```
declare
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 for the procedural definition. 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 semantics to the procedure call `{NRev Xs Ys}`.

[1] Including `Append` and `NRev`, which are bound to procedure values (lexically-scoped closures).

From the semantics outlined above, it follows that `Append` and `NRev` do not search. If there is not enough information to continue, then the computation will simply block. For example, take these two calls:

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

(The **`declare`** ... **`in`** introduces new variables.) The first call, `{Append [1] X Y}`, will run to completion since `Append` does not need the value of its second argument. The result is the binding of `Y` to `1|X`. The second call, `{Append A [2] B}`, will suspend the thread it is executing in. This is because the **`case`** statement does not have enough information to decide what `A` is. No binding is done. If another thread binds `A`, then execution will continue.

This is how Oz supports deterministic logic programming. It is purely declarative logic programming with an operational semantics that is fully specified and deterministic. Programs can be translated in a straightforward way to a Horn clause syntax. However, deductions are not performed by resolution. The execution can be seen as functional programming with logic variables and dynamic typing, carefully designed to have a logical semantics. Resolution was originally designed as an inference rule for automatic theorem provers (Robinson 1965); it is not a necessary part of a logic programming language.

Note that there are higher-order procedures as in a functional language, but no higher-order logic programming, i.e., no logic programming based on a higher-order logic. Higher-order procedures are useful within first-order logic programming as a tool to structure programs and build abstractions.

We find that adding logic variables to functional programming is an important extension for three reasons. First, it allows to do deterministic logic programming in a straightforward way. Second, it increases expressiveness by allowing powerful programming techniques based on incomplete data structures, such as tail-recursive append and difference lists (Clark and Tärnlund 1977; Sterling and Shapiro 1986). The third reason is perhaps the most important: adding concurrency to this execution model gives a useful form of concurrent programming called *declarative concurrency* (see Section 7.2).

3 Nondeterministic logic programming

We call *nondeterministic* logic programming the situation when *search* is used to provide completeness. Using search allows finding solutions when no other algorithm is known.[2] Oz provides the **`choice`** statement as a simple way to introduce search. The **`choice`** statement creates a choice point for its alternatives.

The **`choice`** statement allows to do Prolog-style generative execution. However,

[2] To be precise, search is a general technique that works for any problem by giving just the problem specification, but it can be impractical because it does brute force exploration of a potentially large space of candidate solutions. Search can be made more efficient by incorporating problem-specific knowledge, e.g., games can be programmed using alpha-beta search.

this style of programming does not scale up to real-world search problems.[3] In our opinion, its primary value is pedagogical and exploratory. That is, it can be used on small examples to explore and understand a problem's structure. With this understanding, a more efficient algorithm can often be designed. When used naively, search will not work on large examples due to search space explosion.

Search is a fundamental part of constraint programming. Many techniques have been devised there to reduce greatly the size of the search space. Section 6 gives a simple example to illustrate some of these techniques.

Here is a nondeterministic naive reverse with **choice**:

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

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

(In this and all further examples, we leave out the **declare** for brevity.) Because this example does not use higher-order programming, there is a direct translation to the Horn clause syntax of Prolog:

```
append(Xs, Ys, Zs) :- Xs=nil, Zs=Ys.
append(Xs, Ys, Zs) :- Xs=[X|Xr], Zs=[X|Zr], append(Xr, Ys, Zr).

nrev(Xs, Ys) :- Xs=nil, Ys=nil.
nrev(Xs, Ys) :- Xs=[X|Xr], nrev(Xr, Yr), append(Yr, [X], Ys).
```

If the Oz program is run with depth-first search, its semantics will be identical to the Prolog version.

Controlling search

The program for nondeterministic naive reverse can be called in many ways, e.g., by lazy depth-first search (similar to a Prolog top level) [4], eager search, or interactive search (with the Explorer tool (Schulte 1999b; Van Roy 1999a)). All of these search abilities are programmed in Oz using the notion of computation space (see Section 7). Often the programmer will never use spaces directly (although he or she can), but will use one of the many predefined search abstractions provided in the `Search` module (see Section 6.2).

[3] For problems with a small search space, they may be sufficient. For example, a practical diagnostics generator for the VLSI-BAM microprocessor was written in Prolog (Holmer et al. 1996; Van Roy 1989b).

[4] Lazy search is different from lazy evaluation in that the program must explicitly request the next solution (see Section 7).

As a first example, let us introduce an abstraction, called *search object*, that is similar to a Prolog top level. It does depth-first search and can be queried to obtain successive solutions. Three steps are needed to use it: [5]

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

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

% 3. Calculate and display the first solution:
%    (and others, when repeated)
local X in {E next(X)} {Browse X} end
```

Let us explain each of these steps:

1. The procedure `P` defines the query and returns the solution `S` in its single argument. Because Oz is a higher-order language, the query can be any statement. In this example, the solution has two parts, `X` and `Y`. We pair them together in the tuple `sol(X Y)`.
2. The search object is an instance of the class `Search.object`. The object is created with `New` and initialized with the message `script(P)`.
3. The object invocation `{E next(X)}` finds the next solution of the query `P`. If there is a solution, then `X` is bound to a list containing it as single element. If there are no more solutions, then `X` is bound to `nil`. `Browse` is a tool provided by the system to display data structures.

When running this example, the first call displays the solution `[sol(nil [1 2 3 4 5])]`, that is, a one-element list containing a solution. Successive calls display the solutions `[sol([1] [2 3 4 5])]`, ..., `[sol([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. To solve algorithmic problems, one does not need to learn how to use search in the language. This is unlike Prolog, where search is ubiquitous: even procedure application is defined in terms of resolution, and thus search. In Oz, the **`choice`** statement explicitly creates a choice point, and search abstractions (such as `Search.object`, above) encapsulate and control it. However, the **`choice`** statement 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 whose alternatives depend 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.

[5] For clarity, we leave out syntactic short-cuts. For example, calculating and displaying the next solution can be written as `{Browse {E next($)}}`.

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 a nondeterministic 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 (Haridi and Janson 1990). Oz shares many aspects with AKL but improves over it in particular by being compositional and higher-order.

4.1 Implicit versus explicit concurrency

In early concurrent logic programming systems, concurrency was implicit, driven solely by data dependencies (Shapiro 1989). 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 detecting program termination is hard. 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 suspend. 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 8.2). 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 the 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 list whose tail is an unbound logic variable. The tail can itself be bound to a stream, and so forth.

```
proc {Generate N Limit Xs}
   if N<Limit then Xr in
      Xs=N|Xr
      {Generate N+1 Limit Xr}
   else Xs=nil end
end

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

local Xs S in
   thread {Generate 0 150000 Xs} end  % Producer thread
   thread {Sum Xs 0 S} 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.

This example has exactly one producer feeding exactly one consumer. It therefore does not need a nondeterministic choice. More general cases do, e.g., a client-server application with more than one client feeding a server. Without additional information, the server never knows which client will send the next request. Nondeterministic choice can be added directly to the language, e.g., the `WaitTwo` operation of Section 7.2. It turns out to be more practical to add state instead. Then nondeterministic choice is a consequence of having both state and concurrency, as explained in Section 5.

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 *explicit 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 explicit triggers. The consumer binds the end of a stream to `X|_`. The producer waits for this and binds `X` to the next list element.

Explicit triggers are cumbersome because they require 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 syntactically by annotating the function as "`lazy`". Here is 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

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

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

Sequential systems often support coroutining as a simple way to get some of the abilities of concurrency. Coroutining is a form of non-preemptive concurrency in which a single locus of control is switched manually between different parts of a program. In our experience, a system with efficient preemptive concurrency almost never needs coroutining.

Most modern Prolog systems support coroutining. The coroutining is either supported directly, as in IC-Prolog (Clark and McCabe 1979; Clark et al. 1982), 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 (Colmerauer 1982), sets up the system to invoke the goal `G` when the variable `X` is bound (Sterling and Shapiro 1986). With `freeze` it is possible to have "non-preemptive 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 Explicit state

From a theoretical point of view, explicit state has often been considered a forbidden fruit in logic programming. We find that using explicit state is important for fundamental reasons related to program modularity (see Chapter 4 of (Van Roy and Haridi 2002)).

There exist tools to use state in Prolog while keeping a logical semantics when possible. See for example SICStus Objects (Carlsson *et al* 1999), Prolog++ (Moss 1994), and the Logical State Threads package (Kågedal et al. 1997). An ancestor of the latter was used to help write the Aquarius Prolog compiler (Van Roy 1989a; Van Roy and Despain 1992).

Functional programmers have also incorporated state into functional languages, e.g., by means of `set` operations in LISP/Scheme (Steele, Jr. 1984; Abelson et al. 1996), references in ML (Milner et al. 1990), and monads in Haskell (Wadler 1992).

5.1 Cells (mutable references)

State is an explicit part of the basic execution model in Oz. The model defines the concept of *cell*, which is a kind of mutable reference. A cell is a pair of a name `C` and a reference `X`. There are two operations on cells:

```
{NewCell X C}     % Create new cell with name C and content X
{Exchange C X Y} % Update content to Y and bind X to old content
```

Each `Exchange` atomically accesses the current content and defines a new content.

Oz has a full-featured concurrent object system which is completely defined in terms of cells (Henz 1997b; Henz 1997a). The object system includes multiple inheritance, fine-grained method access control, and first-class messages. Sections 7 gives more information about cells and explains how they underlie the object system.

5.2 Ports (communication channels)

In this section we present another, equivalent way to add state to the basic model. This is the concept of *port*, which was pioneered by AKL. A port is a pair of a name `P` and a stream `Xs` (Janson et al. 1993). There are two operations on ports:

```
{NewPort Xs P}  % Create new port with name P and stream Xs
{Send P X}      % Add X to port's stream asynchronously
```

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 (Armstrong et al. 1996). Erlang extends it by adding to each object a mailbox that does retrieval by pattern matching.

With cells it is natural to define non-active objects, called *passive objects*, shared between threads. With ports it is natural to define active objects that send messages to each other. From a theoretical point of view, these two programming styles have the same expressiveness, since cells and ports can be defined in terms of each other without changing time or space complexity (Henz 1997a; Lauer and Needham 1978). They differ in practice, since depending on the application one style might be more convenient than the other. Database applications, which are centered around a shared data repository, find the shared object style natural. Multi-agent applications, defined in terms of collaborating active entities, find the active object style natural.

5.3 Relevance to concurrent logic programming

From the perspective of concurrent logic programming, explicit state amounts to the addition of a constant-time n-way stream merge, where n can grow arbitrarily large at run-time. That is, any number of threads can concurrently send to the same port, and each send will take constant time. This can be seen as the ability to give an *identity* to an active object. The identity is a first-class value: it can be stored in a data structure and can be passed as an argument. It is enough to know the identity to send a message to the active object.

Without explicit state it impossible to build this kind of merge. If n is known only at run-time, the only solution is 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.

5.4 Creating an active object

Here is 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
```

The notation **proc** `{$ X} ...` **end** defines an *anonymous* procedure value, which is not bound to any identifier. `ForAll` is a higher-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 **choice** statement and the lazy depth-first abstraction `Search.object`. The programming style shown there is too limited for many realistic problems. This section shows how to make search more practical in Oz. We only scratch 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 (Schulte and Smolka 1999; Müller 1999).

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 (Sterling and Shapiro 1986)):

```
proc {Father F C}
   choice 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
```

Now consider the following Prolog predicate:

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

This is defined in Oz as follows:

```
proc {ChildrenFun X Kids}
F in
   proc {F K} {Father X K} end
   {Search.base.all F Kids}
end
```

The procedure `F` is a lexically-scoped closure: it has the external reference `X` hidden inside. This can be written more compactly with an anonymous procedure value:

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

The `Search.base.all` abstraction takes a one-argument procedure and returns the list of all solutions to the procedure. The example call:

```
{Browse {ChildrenFun terach}}
```

returns `[abraham nachor haran]`. The `ChildrenFun` definition is deterministic; if called with a known `X` then it returns `Kids`. To search over different values of `X` we give the following definition instead:

```
proc {ChildrenRel 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 tiny scope. The example call:

```
{Browse {Search.base.all
  proc {$ Q} X Kids in {ChildrenRel X Kids} Q=sol(X Kids) end}}
```

returns:

```
[sol(terach [abraham nachor haran])
 sol(terach [abraham nachor haran])
 sol(terach [abraham nachor haran])
 sol(abraham [isaac])
 sol(haran [lot milcah yiscah])
 sol(haran [lot milcah yiscah])
 sol(haran [lot milcah yiscah])]
```

In Prolog, `bagof` can use existential quantification. For example, the Prolog predicate:

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

collects all children such that there exists a father. This 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" notation `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 (Duchier et al. 1999). All procedures take as argument a unary procedure `{P X}`, where `X` is bound to a solution. Except for lazy search, they all provide depth-first search (one and all solution) and branch-and-bound search (with a cost function). Here are the procedures:

- **Basic search**. This is the simplest to use; no extra parameters are needed.
- **General-purpose search**. This allows parameterizing 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 using strategies combining cloning and recomputation is competitive in time and memory with systems using trailing (Schulte 1999a). 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 with linear speedups (Schulte 2002; Schulte 2000b; Schulte 2000a). The order in which the search tree is explored is nondeterministic, and is likely to be different from depth-first or breadth-first. If the entire tree is explored, then the number of exploration steps is the same as depth-first search. The speedup is a consequence of this fact together with the spreading of work.
- **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 (Schulte 1999b; Schulte 1997a). 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 or her own.

6.3 A more scalable way to do search

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

```
functor Fractions   % Name of module specification
   import FD        % Needs the module FD
   export script:P  % Procedure P defines the problem
define
   proc {P Sol}
      A B C D E F G H I BC EF HI
   in
      Sol=sol(a:A b:B c:C d:D e:E f:F g:G h:H i:I)
      BC={FD.decl} EF={FD.decl} HI={FD.decl}
      %%% 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 distribution strategy:
      {FD.distribute ff Sol}
   end
end
```

Fig. 1. A more scalable way to do search

- 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 deductions over big parts of the problem (i.e., deductions that consider many problem variables together). 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).[6] See, e.g., (Smolka 1996), for more explanation.

This approach is widely applicable. For example, it is being applied successfully to computational linguistics (Duchier 1999; Koller and Niehren 2000; Duchier et al. 1999). 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:

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

[6] The term "distribution" as used here refers to the distribution of $\wedge$ over $\vee$ in the logical formula $c \wedge (a \vee b)$ and has nothing to do with distributed systems consisting of independent computers connected by a network.

Here, BC represents the integer $10 \times B + C$. Figure 1 shows how to specify this as a constraint problem. The unary procedure `{P Sol}` fully defines the problem and the distribution strategy. The problem is specified as a conjunction of constraints on `Sol`, which is bound to a record that contains the solution.[7] The record has fields `a`, ..., `i`, one for each solution variable. The problem constraints are expressed in terms of *finite domains*, i.e., finite sets of integers. For example, the notation `1#9` represents the set $\{1, 2, \cdots, 9\}$. The constraints are defined in the module `FD` (Duchier et al. 1999). For example, `FD.distinct` is a global constraint that asserts that all its component variables are distinct integers.

`Fractions` defines `P` inside a *functor*, i.e., a module specification, in Oz terminology. 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 (Duchier et al. 1998). If execution is always in the same process, then the functor is not needed and it is enough to define the procedure `P`. 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 the three machines `adventure`, `galley`, and `norge`. The engine is implemented using computation spaces (see Section 7.4) and Mozart's support for distributed computing (see (Haridi et al. 1998)). A single process is created on each of these machines using the remote shell operation `rsh` (other operations are possible including secure shell `ssh` for secure communication and local shell `sh` for shared-memory multiprocessors). The following command does parallel search on the problem specified in `Fractions`:

```
local X in {E all(Fractions X)} {Browse X} end
```

This installs the functor `Fractions` on each of the three machines and generates all the solutions. This is an example of a more scalable way to do search: first 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 (Caseau et al. 1999; Laburthe and Caseau 1998; Caseau et al. 1999) and OPL (Van Hentenryck 1999). Search is also an important part of constraint programming in general (Marriott and Stuckey 1999).

7 The Oz execution model

So far, we have highlighted different parts of Oz without showing how they interact, something like the proverbial elephant that is different things to different 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

[7] To be precise, `Sol` is bound to a feature tree, which is a logical formulation of a record.

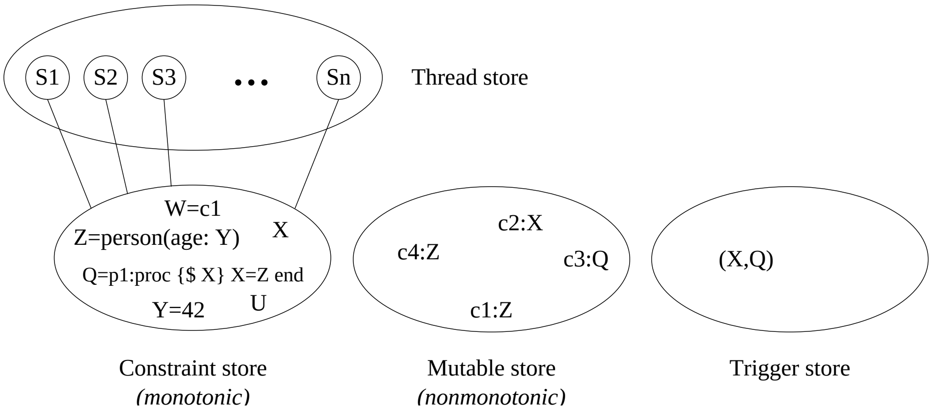

Fig. 2. The Oz store.

multiple paradigms is useful. Finally, Section 7.4 defines computation spaces and how they are used to program search.

7.1 The store

The Oz store consists of four parts (see Figure 2): a thread store, a constraint store, a mutable store, and a trigger store. The constraint store contains equality constraints over the domain of rational trees. 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 references into the constraint store. Mutable references are also called *cells* (Henz 1997a). A mutable reference 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 a mutable reference can be changed.

The trigger store consists of *triggers*, which are pairs of variables and one-argument procedures. Since these triggers are part of the basic execution model, they are sometimes called *implicit triggers*, as opposed to the explicit triggers of Section 4.3. Triggers implement by-need computation (i.e., lazy execution) and are installed with the `ByNeed` operation. We will not say much about triggers in this article. For more information, see (Van Roy and Haridi 2002; Mehl et al. 1998).

The thread store consists of a set of threads. Each thread is defined by a statement S_i. Threads can only have references in the constraint store, not into the other stores. This means that the only way for threads to communicate and synchronize is through shared references in the constraint store. We say a thread is *runnable*, also called *ready*, if it can execute its statement. Threads are *dataflow* threads,

```
⟨s⟩ ::= skip
      | ⟨x⟩1=⟨x⟩2
      | ⟨x⟩=⟨l⟩(⟨f⟩1:⟨x⟩1 ... ⟨f⟩n:⟨x⟩n)
      | ⟨s⟩1 ⟨s⟩2
      | local ⟨x⟩ in ⟨s⟩ end
      | if ⟨x⟩ then ⟨s⟩1 else ⟨s⟩2 end
      | case ⟨x⟩ of ⟨l⟩(⟨f⟩1:⟨x⟩1 ... ⟨f⟩n:⟨x⟩n) then ⟨s⟩1 else ⟨s⟩2 end
      | proc {⟨x⟩ ⟨y⟩1 ... ⟨y⟩n} ⟨s⟩ end
      | {⟨x⟩ ⟨y⟩1 ... ⟨y⟩n}                                    CORE

      | thread ⟨s⟩ end                                         CONCURRENCY

      | {ByNeed ⟨x⟩ ⟨y⟩}                                       LAZINESS

      | try ⟨s⟩1 catch ⟨x⟩ then ⟨s⟩2 end
      | raise ⟨x⟩ end                                          EXCEPTIONS

      | {NewName ⟨x⟩}                                          SECURITY

      | {IsDet ⟨x⟩ ⟨y⟩}
      | {NewCell ⟨x⟩ ⟨y⟩}
      | {Exchange ⟨x⟩ ⟨y⟩ ⟨z⟩}                                 STATE

      | ⟨space⟩                                                SEARCH
```

Fig. 3. The Oz kernel language.

i.e., a thread becomes runnable when the arguments needed by its statement are bound. If an argument is unbound then the thread automatically suspends until the argument is bound. Since the constraint store is monotonic, a thread that is runnable will stay runnable at least until it executes one step of its statement. The system guarantees weak fairness, which implies that a runnable thread will eventually execute.

7.2 The kernel language

All Oz execution can be defined in terms of a simple kernel language, whose syntax is defined in Figure 3. 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. This performance optimization can be seen as a second kernel language, in between full Oz and the kernel language. The second kernel language is implemented directly.

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 3, statements are denoted by ⟨s⟩, computation space operations by ⟨space⟩ (see Figure 6), logic variables by ⟨x⟩, ⟨y⟩, ⟨z⟩, record labels by ⟨l⟩, and record field names by ⟨f⟩.

The semantics of the kernel language is given in (Van Roy and Haridi 2002) (except for spaces) and (Schulte 2002; Schulte 2000b) (for spaces). For comparison, the semantics of the original Oz language is given in (Smolka 1995a). The kernel language splits naturally into seven 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. The **`local`** statement introduces new variables (**`declare`** is a syntactic variant whose scope extends over the whole program).
- *CONCURRENCY*: The concurrency support adds explicit thread creation. Together with the core, this gives *dataflow concurrency*, which is a form of declarative concurrency. Compared to a sequential program, this gives the same results but incrementally instead of all at once. This is deterministic logic programming with more flexible control than the core alone. This is discussed at length in (Van Roy and Haridi 2002).
- *LAZINESS*: The laziness support adds the `ByNeed` operation, which allows to express lazy execution, which is the basic idea of nonstrict functional languages such as Haskell (Mehl et al. 1998; Mehl 1999; Hudak et al. 1992).[8] Together with the core, this gives *demand-driven concurrency*, which is another form of declarative concurrency. Lazy execution gives the same results as eager execution, but calculates only what is needed to achieve the results. Again, this is deterministic logic programming with more flexible control than the core alone. This is important for resource management and program modularity. Lazy execution can give results in cases when eager execution does not terminate.
- *EXCEPTIONS*: The exception-handling support adds an operation, **`try`**, to create an exception context and an operation, **`raise`**, to jump to the innermost enclosing exception context.
- *SECURITY*: The security support adds *name values*, which are unforgeable constants that do not have a printable representation. Calling `{NewName X}` creates a fresh name and binds it to `X`. A name is a first-class "right" or "key" that supports many programming techniques related to security and encapsulation.
- *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 sufficient for sequential object-oriented programming (Smolka 1995b; Henz 1997b; Henz 1997a). Another, equivalent way to add state is by means of ports, which are explained in Section 5.
- *SEARCH*: The search support adds operations on computation spaces (shown as ⟨space⟩), which are explained in Section 7.4. This allows to express nondeterministic logic programming (see Sections 3 and 6). A computation space encapsulates a choice point, i.e., don't-know nondeterminism, allowing the

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

program to decide how to pick alternatives. Section 7.4 explains spaces in more detail and shows how to program search with them. The **`choice`** statement, which is used in the examples of Sections 3 and 6.1, can be programmed with spaces (see Section 7.4.5).

7.2.1 Concurrency and state

Adding both concurrency and state to the core results in the most expressive computation model. There are two basic approaches to program in it: message passing with active objects or atomic actions on shared state. Active objects are used in Erlang (Armstrong et al. 1996). Atomic actions are used in Java and other concurrent object-oriented languages (Lea 2000). These two approaches have the same expressive power, but are appropriate for different classes of applications (multi-agent versus data-centered) (Van Roy and Haridi 2002; Lauer and Needham 1978).

7.2.2 Nondeterministic choice

Concurrent logic programming is obtained by extending the core with concurrency and nondeterministic choice. This gives a model that is more expressive than declarative concurrency and less expressive than concurrency and state used together. Nondeterministic choice means to wait concurrently for one of several conditions to become true. For example, we could add the operation `WaitTwo` to the core with concurrency. `{WaitTwo X Y}` blocks until either `X` or `Y` is bound to a nonvariable term.[9] It then returns with 1 or 2. It can return 1 if `X` is bound and 2 if `Y` is bound. `WaitTwo` does not need to be added as an additional concept; it can be programmed in the core with concurrency and state.

7.2.3 Lazy functions

The `lazy` annotation used in Section 4.3 is defined in terms of `ByNeed`. Calling `{ByNeed P X}` adds the trigger `(X,P)` to the trigger store. This makes `X` behave as a read-only variable. Doing a computation that needs `X` or attempts to bind `X` will block the computation, execute `{P Y}` in a new thread, bind `Y` to `X`, and then continue. We say a value is *needed* by an operation if the thread executing the operation would suspend if the value were not present. For example, the function:

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

is defined as:

[9] In Mozart, the module `Record` contains this operation: `{WaitTwo X Y}` is written as `{Record.waitOr X#Y}`.

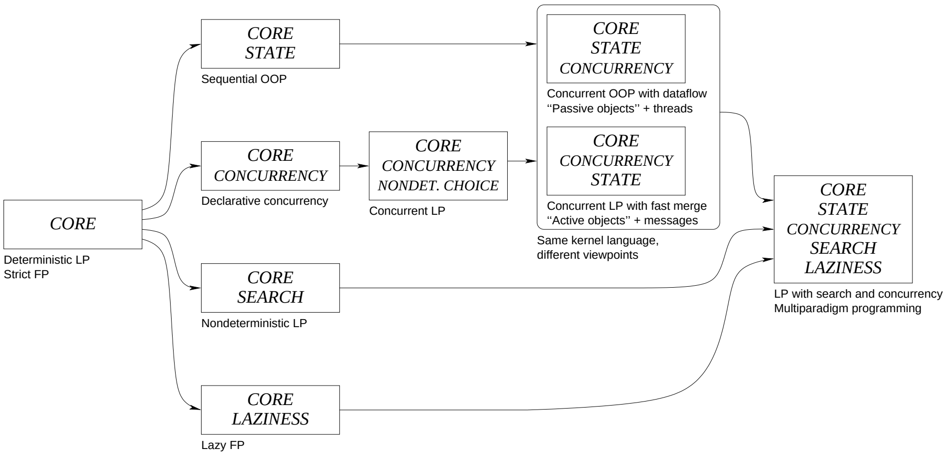

Fig. 4. Some programming paradigms in Oz.

```
fun {Generate N}
P X in
   proc {P Y} Y=N|{Generate N+1} end
   {ByNeed P X}
   X
end
```

`P` will only be called when the value of `{Generate N}` is needed. We make two comments about this definition. First, the `lazy` annotation is given explicitly by the programmer. Functions without it are eager. Second, Mozart threads are extremely lightweight, so the definition is practical. This is a different approach than in nonstrict languages such as Haskell, where lazy evaluation is the default and strictness analysis is used to regain the efficiency of eager evaluation (Hudak et al. 1992).

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 deduction and search), and sequential programming (programming with a totally-ordered sequence of instructions). Some interesting subsets of the kernel language are shown in Figure 4. 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.

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 (see Section 9.4) (Schlichting and Thomas 1991; Van Roy and Haridi 2002). Even if the same basic functionality is provided, it may be useful to view it according to different paradigms depending on the application needs (Lauer and Needham 1978).

How is it possible for such a simple kernel language to support such different programming styles? It is because paradigms have many concepts in common, as Figures 3 and 4 show. A good example is sequential object-oriented programming, which can be built from the core by adding just state (see (Smolka 1995b) 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.

Oz has syntactic support to make this style easy to use and implementation support to make it efficient. The same applies to the declarative paradigms of functional and logic programming. Strict functions are restricted versions of procedures in which the binding is directional. 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 true within the core. It remains true if one adds concurrency and laziness to the core. We illustrate the logical semantics with many examples in this article, starting in Section 2. In the core, the `if` and `case` statements have a logical semantics, i.e., they check entailment and disentailment. 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 by means of computation spaces. When combined with the rest of the model, they make it possible to program a wide variety of search algorithms in Oz, as explained in the next section.

7.4 Computation spaces

Computation spaces are a powerful 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 are a natural way to integrate search into a concurrent system. Spaces 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 (Schulte 1999a). Spaces are compositional, i.e., they can be nested, which is important for building well-structured programs.

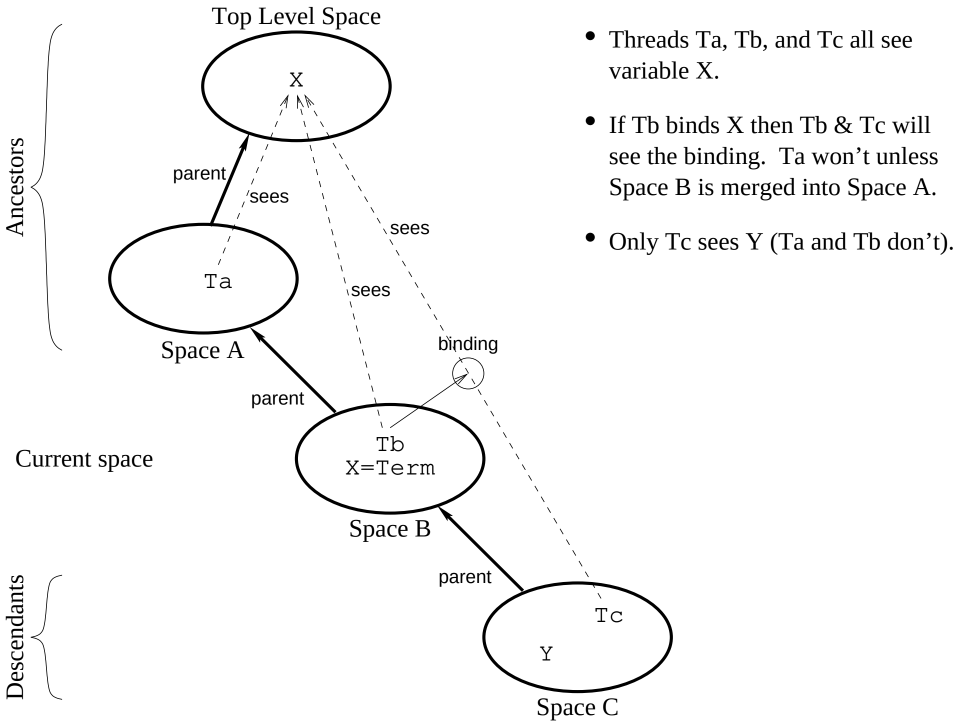

Fig. 5. Visibility of variables and bindings in nested spaces.

This section defines computation spaces, the operations that can be performed on them (see Figure 6), and gives a few examples of how to use them to program search. The discussion in this section follows the model in (Schulte 2002; Schulte 2000b). This model is implemented in Mozart 1.1.0 (Mozart Consortium 2000) and refines the one presented in the articles (Schulte 1997b; Schulte 2000c). The space abstraction can be made language-independent; (Henz et al. 1999) describes a C++ implementation of a similar abstraction that supports both trailing and copying.

7.4.1 Definition

A computation space is just an Oz store with its four parts. The store we have seen so far is a single computation space with equality constraints over rational trees. To deal with search, we extend this in two ways. First, we allow spaces to be nested. Second, we allow other constraint systems in a space. Since spaces are used to encapsulate potential variable bindings, it is important to be precise about the visibility of variables and bindings. Figure 5 gives an example. The general rules for the structure of computation spaces are as 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 threads in the space and its descendants, 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.
  - 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; attempting to create another gives a run-time error.
  - 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 its parent. Any further operation on the space is an error. This state is the end of a space's lifetime.

- The space is *suspended*. This means that additional bindings done in an ancestor can make the space runnable. Being suspended 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 suspended indefinitely usually indicates a programmer error.

7.4.3 Programming search

A *search strategy* defines how the search tree is explored, e.g., depth-first seach, limited discrepancy search, best-first search, and branch-and-bound search. A *distribution strategy* defines the shape and content of the search tree, i.e., how many alternatives exist at a node and what constraint is added for each alternative. Computation spaces can be used to program search strategies and distribution strategies independent of each other. That is, any search strategy can be used together with any distribution strategy. Here is how it is done:

- Create the space and initialize it by running an internal program that defines all the variables and constraints in the space.
- Propagate information inside the space. The constraints in a space have an operational semantics. In Oz terminology, an operationalized version of a constraint is called a *propagator*. Propagators execute concurrently; each propagator executes inside its own thread. Each propagator reads its arguments and attempts to add information to the constraint store by restricting the domains of its arguments.
- All propagators execute until no more information can be added to the store in this manner. This is a fixpoint calculation. When no more information can be added, then the fixpoint is reached and the space has become stable.
- During a space's execution, the computation inside the space can decide to create a choice point. The decision which constraint to add for each alternative defines the distribution strategy. One of the space's threads will suspend when the choice point is created.
- When the space has become stable, then execution continues outside the space, to decide what to do next. There are different possibilities depending on whether or not a choice point has been created in the space. If there is none, then execution can stop and return with a solution. If there is one, then the search strategy decides which alternative to choose and commits to that alternative.

Notice that the distribution strategy is problem-dependent: to add a constraint we need to know the problem's constraints. On the other hand, the search strategy is problem-independent: to pick an alternative we do not need to know which constraint it corresponds to. The next section explains the operations we need to implement this approach. Then, Section 7.4.5 gives some examples of how to program search.

```
⟨space⟩ ::=  {NewSpace ⟨x⟩ ⟨y⟩}
          |  {Choose ⟨x⟩ ⟨y⟩}
          |  {Ask ⟨x⟩ ⟨y⟩}
          |  {Commit ⟨x⟩ ⟨y⟩}
          |  {Clone ⟨x⟩ ⟨y⟩}
          |  {Inject ⟨x⟩ ⟨y⟩}
          |  {Merge ⟨x⟩ ⟨y⟩}
```

Fig. 6. Primitive operations for computation spaces.

7.4.4 Space operations

Now we know enough to define the primitive space operations. There are seven principal ones (see Figure 6).

- `{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 `{P R}` is invoked in a new thread.
- `{Choose N Y}` is the only operation that executes *inside* a space. It creates 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 may exist in a space at any time.
- `{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 blocked `Choose` call in the space to continue with `I` as its result. This may cause a stable space to become not stable again. The space will resume execution until a new fixpoint is reached. 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 the alternatives of a distributable space to be explored independently.
- `{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 may cause a stable space to become not stable again. The space will resume execution until a new fixpoint is reached. 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.

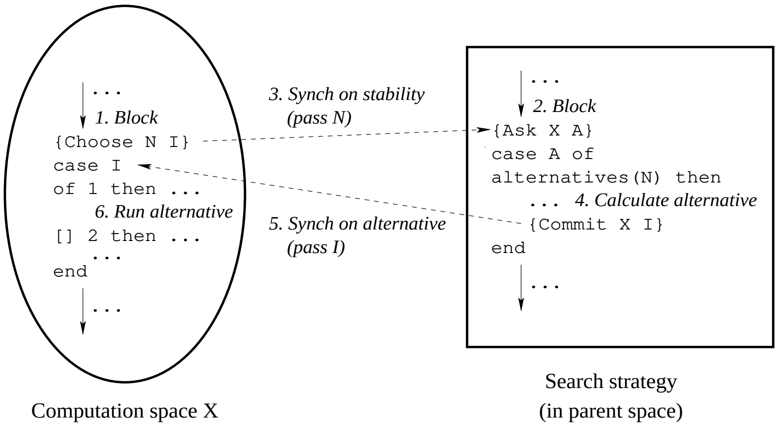

Fig. 7. Communication between a space and its search strategy.

7.4.5 Using spaces

These seven primitive operations are enough to define many search strategies and distribution strategies. The basic technique is to use `Choose`, `Ask`, and `Commit` to communicate between the inside of the space and the outside of the space. Figure 7 shows how the communication works: first the space informs the search strategy of the total number of alternatives (`N`). Then the search strategy picks one (`I`) and informs the space. Let us now present briefly a few examples of how to use spaces. For complete information on these examples and many other examples we refer the reader to (Schulte 2002; Schulte 2000b).

Depth-first search. Our first example implements a search strategy. Figure 8 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 `{P 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 `P`, choice points are defined with the `Choose` operation.

Naive choice point. Our second example implements a distribution strategy. Let us implement a naive choice point, namely one that defines a set of alternative statements to be chosen. This can be defined as follows:

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

Fig. 8. Depth-first single solution search.

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

Oz provides the following more convenient syntax for this technique:

```
choice S1 [] ... [] Sn end
```

This is exactly how the **choice** statement is defined. This statement can be used with any search strategy, such as the depth-first strategy we defined previously or other strategies.

Andorra-style disjunction (the **dis** *statement).* Let us now define a slightly more complex distribution strategy. We define the **dis** statement, which is an extension of **choice** that eliminates failed alternatives and commits immediately if there is a single remaining alternative:

```
dis G1 then S1 [] ... [] Gn then Sn end
```

In contrast to **choice**, each alternative of a **dis** statement has both a guard and a body. The guards are used immediately to check failure. If a guard G_i fails then its alternative is eliminated. This extension is sometimes called determinacy-directed execution. It was discovered by D.H.D. Warren and called the *Andorra principle* (Haridi and Brand 1988; Santos Costa et al. 1991).

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. See (Schulte 2002; Schulte 2000b) for details of the implementation. It can be optimized to do first-argument indexing in a similar way to Prolog systems. We emphasize that the whole implementation is written within the language.

The first-fail strategy. In practice, **`dis`** is not strong enough for solving real constraint problems. It is too static: its alternatives are defined textually in the program code. A more sophisticated distribution strategy would look more closely at the actual state of the execution. For example, the *first-fail* strategy for finite domain constraints looks at all variables and places a choice point on the variable whose domain is the smallest. First-fail can be implemented with `Choose` and a set of reflective operations on finite domain constraints. The Mozart system provides first-fail as one of many preprogrammed strategies.

Deep guard combinators. A *constraint combinator* is an operator that takes constraints as arguments and combines them to form another constraint. Spaces are a powerful way to implement constraint combinators. Since spaces are compositional, the resulting constraints can themselves be used as inputs to other constraint combinators. For this reason, these combinators are called *deep guard* combinators. This is more powerful than other techniques, such as reification, which are *flat*: their input constraints are limited to simple combinations of built-in constraints. Some examples of deep guard combinators that we can program are deep negation, generalized reification, propagation-based disjunction (such as **`dis`**), constructive disjunction, and deep committed-choice.

8 Related work

We first give a brief overview of research in the area of multiparadigm programming. We then give a short history of Oz.

8.1 Multiparadigm languages

Integration of paradigms is an active area of research that has produced a variety of different languages. We give a brief glimpse into this area. We do not pretend to be exhaustive; that would be the subject of another paper. As far as we know, there is no other language that covers as many paradigms as Oz in an equitable way, i.e., with a simple formal semantics (Smolka 1995a; Van Roy and Haridi 2002) and an efficient implementation (Mehl et al. 1995; Mehl 1999; Scheidhauer 1998; Schulte 2002; Schulte 2000b). An early discussion of multiparadigm programming in Oz is given in (Martin Müller et al. 1995). It gives examples in functional, logic, and object-oriented styles.

A short-term solution to integrate different paradigms is to use a *coordination model* (Carriero and Gelernter 1989; Carriero and Gelernter 1992). The prototypical coordination model is Linda, which provides a uniform global tuple space that can be accessed with a small set of basic operations (concurrent reads and writes) from any process that is connected to it. A Linda layer can act as "glue" between languages of different paradigms. Let us now look at more substantive solutions.

Within the imperative paradigm, there have been several efforts to add the abilities of functional programming. Smalltalk has "blocks", which are lexically-scoped closures (Goldberg and Robson 1983). Java has inner classes, which (with minor limitations) are lexically-scoped closures. Java supports the `final` annotation, which allows programming with stateless objects. Using inner classes and `final` allows to do functional programming in Java. However, this technique is verbose and its use is discouraged (Arnold and Gosling 1998). More ambitious efforts are C++ libraries such as FC++ (McNamara and Smaragdakis 2000) and language extensions such as Pizza (Odersky and Wadler 1997) and Brew (Baumgartner et al. 2001), which translate into Java. These provide much better support for functional programming.

Within the functional paradigm, the easiest way to allow imperative programming is to add locations with destructive assignment. This route was taken by languages such as Lisp (Steele, Jr. 1984), Scheme (Clinger and Rees 1991), and SML (Harper et al. 1986). The M-structures of Id (Nikhil 1994a) and its successor pH (Nikhil 1994b; Nikhil and Arvind 2001) fall in this category as well. Objective Caml is a popular object-oriented dialect of ML that takes this approach (Chailloux et al. 2000; Rémy and Vouillon 1998). Oz also takes this approach, building an object system from a functional core by adding the cell as its location primitive.

In Haskell, state is integrated using the monadic style of programming (Wadler 1992; Peyton Jones and Wadler 1993) which generalizes the continuation-passing style. Because Haskell is a nonstrict language, it cannot easily add locations with destructive assignment. The monadic style allows to control the sequentialization necessary for various kinds of side effecting (I/O, error handling, nondeterministic choice). However, because it imposes a global state threading, it has difficulties when integrated with concurrency. See (Van Roy and Haridi 2002) for a discussion of the relative merits of the state threading approach versus the location approach.

Within the logic paradigm, there have been many attempts to add an object system (Davison 1993). Prominent examples are Prolog++ (Moss 1994) and SICStus Objects (Carlsson *et al* 1999). These approaches use locations as primitives, much like the functional approach.

Functions have been added in several ways to logic languages. A first approach is LIFE, which provides functions as a kind of relation that is called by *entailment*, i.e, the function call waits until its arguments have enough information. This delaying mechanism is called *residuation* (Aït-Kaci and Nasr 1989; Aït-Kaci and Podelski 1993; Aït-Kaci and Lincoln 1988; Aït-Kaci et al. 1994). A second approach extends the basic resolution step to include the deterministic evaluation of functions. This execution strategy, called *narrowing*, underlies the Curry language (Hanus 1994; Hanus 1997). A third approach is taken by Lambda Prolog (Nadathur and Miller 1995). It uses a more powerful logic than Horn logic as a basis for programming. In particular, functional programming is supported by providing λ terms as data structures, which are handled by a form of higher-order unification. A fourth approach is taken by HiLog (Chen et al. 1993), which introduces a higher-order syntax that can be encoded into the first-order predicate calculus.

The Oz approach is to provide first-class procedure values and to consider them as constants for the purposes of unification. This approach cleanly separates the logical aspects from the higher-order programming aspects. All the other approaches mentioned are more closely tied to the resolution operation. In addition, the Oz approach provides the full power of lexically-scoped closures as values in the language. Finally, Oz provides entailment checking as a separate operation, which allows it to implement call by entailment.

Erlang is a notable example of a multiparadigm language. It has a layered design (Armstrong et al. 1996; Wikström 1994). Erlang programs consist of active objects that send messages to each other. A strict functional language is used to program the internals of the active objects. Each active object contains one thread that runs a recursive function. The object state is contained in the function arguments. This model is extended further with distribution and fault tolerance.

The layered approach is also taken by pH, a language designed for defining algorithms with implicit parallelism (Nikhil 1994b; Nikhil and Arvind 2001). Its core is based on Haskell. It has two extensions. The first extension is a single-assignment data type, I-structures. This allows to write functional programs that have dataflow behavior. The second extension is a mutable data type, M-structures. This allows stateful programs. This design has similarities to Oz, with logic variables being the single-assignment extension and cells the mutable extension.

Concurrent logic programming has investigated in depth the use of logic variables for synchronization and communication. They are one of the most expressive mechanisms for practical concurrent programming (Bal et al. 1989; Van Roy and Haridi 2002). Since logic variables are constrained monotonically, they can express monotonic synchronization. This allows declarative concurrency, which is concurrent programming with no observable nondeterminism. The concurrent logic language Strand evolved into the coordination language PCN (Foster 1993) for imperative languages. In the functional programming community, the futures of Multilisp (Halstead 1985) and the I-structures of Id (Nikhil 1994a) allow to synchronize on the result of a concurrent computation. Both realize a restricted form of logic variable. Finally, the Goffin project (Chakravarty et al. 1995) uses a first-order concurrent constraint language as a coordination language for Haskell processes.

The multiparadigm language Leda was developed for educational purposes (Budd 1995). It is sequential, supports functional and object-oriented programming, and has basic support for backtracking and a simple form of logic programming that is a subset of Prolog.

8.2 History of Oz

Oz is a recent descendant of a long line of logic-based languages that originated with Prolog (see Figure 9). 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

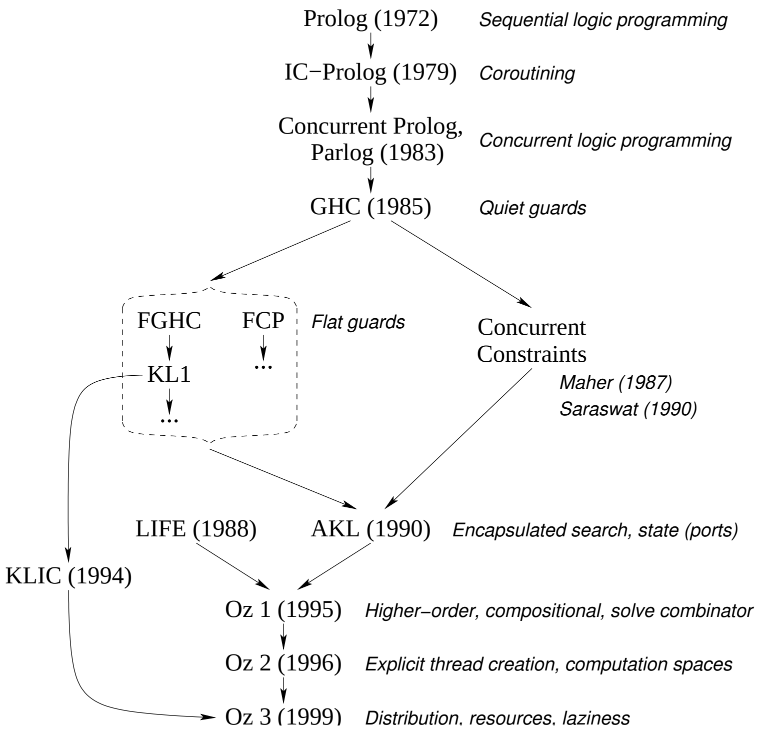

Fig. 9. History of Oz.

concurrent processes (Clark and McCabe 1979; Clark et al. 1982). This led to Parlog and Concurrent Prolog, which introduced the process model of logic programming, usually known as *concurrent* logic programming (Clark 1987; Shapiro 1983; Shapiro 1987). The advent of GHC (Guarded Horn Clauses) simplified concurrent logic programming considerably by introducing the notion of *quiet guards* (Ueda 1985). 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 (Maher 1987; Saraswat and Rinard 1990; Saraswat 1993). The main insight is that logical notions such as equality and entailment can be given an operational reading. Saraswat's concurrent constraint model is a model of concurrent programming with a logical foundation. This model was subsequently used as the basis for several languages including AKL and Oz.

On the practical side, systems with "flat" guards (which are limited to basic constraints or system-provided tests) were the focus of much work (Tick 1995). The flat versions of Concurrent Prolog and GHC, called FCP and FGHC respectively, were developed into large systems (Institute for New Generation Computer Technology 1992; Shapiro 1989). 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 (Fujise et al. 1994). Some of the implementation techniques in the current Mozart system were inspired by KLIC, notably the distributed garbage collection algorithm.

An important subsequent development was AKL (Andorra Kernel Language) (Janson and Haridi 1991; Janson 1994; Janson et al. 1993), which added state (in the form of ports), encapsulated search, and an efficient implementation of deep guards. AKL is the first language that combines the abilities of constraint logic programming and concurrent logic programming. AKL implements encapsulated search using a precursor of computation spaces. When local propagation within a space cannot choose between different disjuncts, then the program can try each disjunct by cloning the computation space.

The initial Oz language, Oz 1, was inspired by AKL and LIFE, and added higher-order procedures, programmable search based on the solve combinator (a less expressive precursor of spaces (Schulte et al. 1994; Schulte and Smolka 1994)), compositional syntax, and the cell primitive for mutable state (Smolka 1995b). Oz 1 features a new record data type that was inspired by LIFE (Smolka and Treinen 1994; Van Roy et al. 1996). Concurrency in Oz 1 is implicit and based on lazy thread creation. When a statement blocks, a new thread is created that contains only the blocked statement. The main thread is not suspended but continues with the next statement. Oz 1 features a concurrent object system designed for lazy thread creation, based on state threading and monitors.

Oz 2 improves on its predecessor Oz 1 with an improved concurrency model and an improved model for encapsulated search. Oz 2 replaces the solve combinator of Oz 1 by computation spaces. In contrast to the solve combinator, spaces allow programming important search strategies such as parallel search, the Oz Explorer, and strategies based on recomputation. 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 system without sequential state threading in 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* (Duchier et al. 1998), and for open, robust, distributed programming (Haridi et al. 1998; Van Roy et al. 1999; Haridi et al. 1999; Van Roy 1999b; Smolka et al. 1995). 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 (Haridi et al. 1999).
- A child computation space is a stationary entity that exists completely in one process. 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 useful for parallel execution. It is optimized to be particularly efficient on shared-memory multiprocessors. For that case, we have experimented with an implementation of interprocess communication using shared pages between address spaces (Haridi et al. 1998).

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 experience, we have learned many lessons both for practical logic programming and for multi-paradigm 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 (demand-driven execution) is useful (Hughes 1989).

9.1 Be explicit ("magic" does not work)

- Provide explicit concurrency (older 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). The majority of Prolog programs solve algorithmic problems, which do not need search, yet one cannot 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.
- Provide explicit state (in C++ and Java, state is implicit, e.g., Java variables are stateful unless declared `final`). By explicit state we mean that the language should declare mutable references only where they are needed. Explicit state should be used sparingly, since it complicates reasoning about programs and is costly to implement in a distributed system. On the other hand, explicit state is crucial for modularity, i.e., the ability to change a program component without having to change other components.
- 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. This allows the system to pay for

laziness only where it is used. A second reason is declarative concurrency: supporting it well requires eager as well as lazy functions. A third reason is explicit state. With implicit laziness (and a fortiori with nonstrictness), it is harder to reason about functions that use explicit state. This is because the order of function evaluation is not determined by syntax but is data dependent.

9.2 Provide primitives for building abstractions

- Full compositionality is essential: everything can be nested everywhere. For maximum usefulness, this requires higher-order procedures with lexical scoping. User-defined abstractions should be carefully designed to be fully compositional.
- The language should be complete enough so that it is easy to define new abstractions. The developer should have all the primitives necessary to build powerful abstractions. 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 (Mehl et al. 1998). 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. This goal is often in conflict with the goal of having an efficient implementation. Satisfying both is difficult, but sometimes possible. One approach that helps is to have a second kernel language, as explained in Section 7.2. Another approach is "loosening and tightening". That is, develop the system in semi-independent stages, where one stage is factored and the next stage brings the factors together. A typical example is a compiler consisting of a naive code generator followed by a smart peephole optimizer.
- It is important to have a sophisticated module system, with lazy loading, support for mutually-dependent modules, and support for application deployment. 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, since modules often depend on each other. Support for application deployment includes the ability to statically link a collection of modules into a single module. This

simplifies how modules are offered to users. A final point is that the module system is written within the language, using records, explicit laziness, and functors implemented by higher-order procedures.
- 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 (Mehl et al. 2000; Müller 2000). These constraint systems are fully integrated into the system, including taking advantage of the full power of computation spaces. The current Mozart system has four constraint systems, based on rational trees (for both "bound records" and "free records" (Van Roy et al. 1996)), finite domains (Schulte and Smolka 1999), and finite sets (Müller 1999). Mozart also supports memory management across the interface, with garbage collection from the Oz side (using finalization and weak pointers) and manual control from 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 often best programmed in different paradigms.[10] 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 and encapsulates the concurrency in the active object.
- Different levels of abstraction are often best expressed in different paradigms. For example, consider a multi-agent system programmed in a concurrent logic language. At the language level, the system does not have the concept of state. But there is 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 language 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. Usually this is not a good idea. We explain why in one particularly interesting case, namely pure concurrent logic programs with state (Janson et al. 1993). 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 the internal stream's most-recent 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,

[10] Another approach is to use multiple languages with well-defined interfaces. This is more complex, but can sometimes work well.

a new stream merger has to be created. The system as a whole is therefore more complex than a system with state:

- The communication 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 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 with identities) is mapped directly to execution model state (e.g., using ports for many-to-one communication), which is compiled directly into machine state. Both the compiler and the run-time system are simple. One may argue that the stateful execution model is no longer "pure". This is true but irrelevant, 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 (Van Roy and Haridi 2002). 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 a programming language should support multiple paradigms.

9.5 Combine dynamic and static typing

We define a *type* as a set of values along with a set of operations on those values. We say that a language has *checked types* if the system enforces that operations are only executed with values of correct types. There are two basic approaches to checked typing, namely dynamic and static typing. In *static typing*, all variable types are known at compile time. No type errors can occur at run-time. In *dynamic typing*, the variable type is known with certainty only when the variable is bound. If a type error occurs at run-time, then an exception is raised. Oz is a dynamically-typed language. Let us examine the trade-offs in each approach.

Dynamic typing puts 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 to test programs or program fragments even when they are in an incomplete or inconsistent state. 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 allows programs, such as operating systems, that run indefinitely and grow and evolve.

On the other hand, static typing has at least three advantages when compared to dynamic typing. It allows to catch more program errors at compile time. It allows for a more efficient implementation, since the compiler can choose a representation appropriate for the type. Last but not least, it allows for partial program verification, since some program properties can be guaranteed by the type checker.

In our experience, we find that neither approach is always clearly better. Sometimes flexibility is what matters; at other times having guarantees is more important. It seems therefore that the right type system should be "mixed", that is, be a combination of static and dynamic typing. This allows the following development methodology, which is consistent with our experience. In the early stages of application development, when we are building prototypes, dynamic typing is used to maximize flexibility. Whenever a part of the application is completed, then it is statically typed to maximize correctness guarantees and efficiency. 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.

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:

- 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.

Our proposal for a mixed type system would go in the opposite direction. In the mixed type system, the default is dynamic typing. Static typing is done as soon as needed, but not before. This means that the trade-off between flexibility and having guarantees is not frozen by the language design, but is made available to the programmer. The design of this mixed type system is a subject for future research.

Mixed typing is related to the concept of "soft typing", an approach to type checking for dynamically-typed languages (Cartwright and Fagan 1991). In soft typing, the type checker cannot always decide at compile time whether the program is correctly typed. When it cannot decide, it inserts run-time checks to ensure safe execution. Mixed typing differs from soft typing in that we would like to avoid the inefficiency of run-time checking, which can potentially change a program's time complexity. The statically-typed parts should be truly statically typed.

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 implementation efficiency in Mozart. The methodology is nowhere fully described in print; there are only partial explanations (Smolka 1995b; Van Roy 1999b). We summarize it here.

At all times during development, there is a robust implementation. However, the system's design is in continuous flux. 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 (Hoare 1987; Ritchie 1987; Thompson 1987). 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*, which consists in developing an initial implementation, exposing it to user comment, and refining it until the system is adequate (Sommerville 1992). 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 double requirement of efficient implementation and 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 (Darwin 1964). In the analogy, the individuals are abstractions and the filters are the two acceptance criteria of efficient implementation and simple 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 nonstrict functional programming, and concurrent object-oriented programming. Oz supports declarative concurrency, a little-known form of concurrent programming that deserves to be more widely known. Because of appropriate syntactic and implementation support, all these paradigms are easy to use. We say that Oz is *multiparadigm*. It is important to be multiparadigm because good program design often requires different paradigms to be used for different parts of a program. To a competent Oz programmer, the conventional boundaries between paradigms are artificial and irrelevant.

The Mozart system implements Oz and is in continuing development by the Mozart Consortium (Mozart Consortium 2000). Research and development started in 1991. The current release has a full-featured development environment and is being used for serious application development. This article covers most of the basic language primitives of Oz. We only briefly discussed the object system, the module system (i.e., functors), and constraint programming, because of space limitations. 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 interfaces. 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 (Smolka et al. 1995). 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 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 De Schreye for suggesting the ICLP99 tutorial on which this article is based. We thank Krzysztof Apt, Manuel Hermenegildo, Kazunori Ueda, and others for their questions and comments at the ICLP99 tutorial where the original talk was given. We thank the anonymous referees for their comments that helped us much improve the presentation. Finally, we give a special thanks to Juris Reinfelds. This research was partly financed by the Walloon Region of Belgium in the PIRATES project.

References

Abelson, H., Sussman, G. J., and Sussman, J. 1996. *Structure and Interpretation of Computer Programs, Second Edition.* The MIT Press, Cambridge, Mass.

Aït-Kaci, H., Dumant, B., Meyer, R., Podelski, A., and Van Roy, P. 1994. The Wild LIFE handbook. Available at `http://www.info.ucl.ac.be/people/PVR/handbook.ps`.

Aït-Kaci, H. and Lincoln, P. 1988. LIFE: A natural language for natural language. MCC Technical Report ACA-ST-074-88, MCC, ACA Program. Feb.

Aït-Kaci, H. and Nasr, R. 1989. Integrating logic and functional programming. *Journal of Lisp and Symbolic Computation 2*, 51–89.

Aït-Kaci, H. and Podelski, A. 1993. Towards a meaning of LIFE. *J. Log. Prog. 16,* 3-4 (July-August), 195–234.

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

Arnold, K. and Gosling, J. 1998. *The Java Programming Language, Second Edition.* Addison Wesley.

Ashenhurst, R. L. and Graham, S., Eds. 1987. *ACM Turing Award Lectures: The First Twenty Years.* ACM Press.

Bal, H. E., Steiner, J. G., and Tanenbaum, A. S. 1989. Programming languages for distributed computing systems. *ACM Comput. Surv. 21,* 3 (Sept.), 261–322.

Baumgartner, G., Jansche, M., and Peisert, C. D. 2001. Support for functional programming in Brew. In *Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL), European Conference on Object-Oriented Programming (ECOOP).* NIC, vol. 7. John von Neumann Institute for Computing, 111–125.

Budd, T. A. 1995. *Multiparadigm Programming in Leda.* Addison-Wesley.

Carlsson *et al*, M. 1999. SICStus Prolog 3.8.1. Available at `http://www.sics.se`.

Carriero, N. and Gelernter, D. 1989. Linda in context. *Commun. ACM 32,* 4, 444–458.

Carriero, N. and Gelernter, D. 1992. Coordination languages and their significance. *Commun. ACM 35,* 2 (Feb.), 96–107.

Cartwright, R. and Fagan, M. 1991. Soft typing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI).* Toronto, Ontario, 278–292.

Caseau, Y., Josset, F.-X., and Laburthe, F. 1999. CLAIRE: Combining sets, search and rules to better express algorithms. In *Proceedings of the 1999 International Conference on Logic Programming (ICLP 99).* The MIT Press, Las Cruces, NM, USA, 245–259.

Caseau, Y., Laburthe, F., and Silverstein, G. 1999. A meta-heuristic factory for vehicle routing problems. In *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming (CP '99)*, J. Jaffar, Ed. Alexandria, VA, USA, 144–158.

Chailloux, E., Manoury, P., and Pagano, B. 2000. *Développement d'applications avec Objective Caml.* O'Reilly, Paris, France. In French.

Chakravarty, M., Guo, Y., and Köhler, M. 1995. Goffin: Higher-order functions meet concurrent constraints. In *First International Workshop on Concurrent Constraint Programming.* Venice, Italy.

Chen, W., Kifer, M., and Warren, D. S. 1993. HiLog: A foundation for higher-order logic programming. *J. Log. Prog. 15,* 3 (Feb.), 187–230.

Clark, K. L. 1987. PARLOG: the language and its applications. In *Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE). Volume II: Parallel Languages*, A. J. N. J. W. de Bakker and P. C. Treleaven, Eds. Lecture Notes in Computer Science, vol. 259. Springer Verlag, Eindhoven, The Netherlands, 30–53.

Clark, K. L. and McCabe, F. 1979. The control facilities of IC-Prolog. In *Expert Systems in the Micro-Electronic Age*, D. Michie, Ed. Edinburgh University Press, Edinburgh, Scotland, 122–149.

Clark, K. L., McCabe, F. G., and Gregory, S. 1982. IC-PROLOG — language features. In *Logic Programming*, K. L. Clark and S.-Å. Tärnlund, Eds. Academic Press, London, 253–266.

Clark, K. L. and Tärnlund, S.-Å. 1977. A first order theory of data and programs. In *Proceedings of the IFIP Congress*. North-Holland, Toronto, Canada, 939–944.

Clinger, W. and Rees, J. 1991. The revised^4 report on the algorithmic language Scheme. *LISP Pointers IV,* 3 (July-September), 1–55.

Colmerauer, A. 1982. PROLOG II reference manual and theoretical model. Tech. rep., Université Aix-Marseille II, Groupe d'Intelligence Artificielle. Oct.

Cousineau, G. and Mauny, M. 1998. *The Functional Approach to Programming*. Cambridge University Press.

Darwin, C. 1964. *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).

Davison, A. 1993. A survey of logic programming-based object oriented languages. In *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press.

Duchier, D. 1999. Axiomatizing dependency parsing using set constraints. In *Sixth Meeting on Mathematics of Language (MOL6)*. Orlando, Florida.

Duchier, D., Gardent, C., and Niehren, J. 1999. Concurrent constraint programming in Oz for natural language processing. Lecture notes, `http://www.ps.uni-sb.de/Papers/abstracts/oznlp.html`.

Duchier, D., Kornstaedt, L., Müller, T., Schulte, C., and Van Roy, P. 1999. System modules. Tech. rep., Mozart Consortium. Jan. Available at `http://www.mozart-oz.org`.

Duchier, D., Kornstaedt, L., Schulte, C., and Smolka, G. 1998. A Higher-order Module Discipline with Separate Compilation, Dynamic Linking, and Pickling. Tech. rep., Programming Systems Lab, DFKI and Universität des Saarlandes. DRAFT.

Foster, I. 1993. Strand and PCN: Two generations of compositional programming languages. Tech. Rep. Preprint MCS-P354-0293, Argonne National Laboratories.

Fujise, T., Chikayama, T., Rokusawa, K., and Nakase, A. 1994. KLIC: A portable implementation of KL1. In *Fifth Generation Computing Systems (FGCS '94)*. 66–79.

Gabbay, D. M., Hogger, C. J., and Robinson, J. A., Eds. 1995. *Handbook of Logic in Artificial Intelligence and Logic Programming*. Vol. 5. Oxford University Press.

Garey, M. R. and Johnson, D. S. 1979. *Computers and Intractibility*. W.H. Freeman and Co.

Goldberg, A. and Robson, D. 1983. *Smalltalk-80: The language and its implementation*. Addison-Wesley.

Halstead, R. H. 1985. MultiLisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst. 7,* 4 (Oct.), 501–538.

Hanus, M. 1994. The integration of functions into logic programming: From theory to practice. *J. Log. Prog. 19/20*, 583–628.

Hanus, M. 1997. A unified computation model for functional and logic programming. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 97)*. Paris, France, 80–93.

Haridi, S. and Brand, P. 1988. Andorra Prolog – an integration of Prolog and committed-choice languages. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS)*, I. of New Generation Computer Technology (ICOT), Ed. Vol. 2. Ohmsha Ltd. and Springer Verlag, Tokyo, Japan, 745–754.

Haridi, S. and Janson, S. 1990. Kernel Andorra Prolog and its computation model. In *Proceedings of the 7th International Conference on Logic Programming (ICLP 90)*. The MIT Press, Jerusalem, Israel, 31–48.

Haridi, S., Van Roy, P., Brand, P., Mehl, M., Scheidhauer, R., and Smolka, G.

1999. Efficient logic variables for distributed computing. *ACM Trans. Program. Lang. Syst. 21,* 3 (May), 569–626.

Haridi, S., Van Roy, P., Brand, P., and Schulte, C. 1998. Programming languages for distributed applications. *New Generation Computing 16,* 3 (May), 223–261.

Harper, R., MacQueen, D., and Milner, R. 1986. Standard ML. Tech. Rep. ECS-LFCS-86-2, University of Edinburgh, Dept. of Computer Science.

Henz, M. 1997a. *Objects for Concurrent Constraint Programming.* The Kluwer International Series in Engineering and Computer Science, vol. 426. Kluwer Academic Publishers, Boston.

Henz, M. 1997b. Objects in Oz. Ph.D. thesis, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany.

Henz, M., Müller, T., and Ng, K. B. 1999. Figaro: Yet another constraint programming library. In *Workshop on Parallelism and Implementation Technology for Constraint Logic Programming, International Conference on Logic Programming (ICLP 99).* Las Cruces, NM, USA.

Hoare, C. A. R. 1987. The emperor's old clothes. See Ashenhurst and Graham (1987). 1980 Turing Award Lecture.

Holmer, B. K., Sano, B., Carlton, M., Roy, P. V., and Despain, A. M. 1996. Design and analysis of hardware for high performance Prolog. *J. Log. Prog. 29*, 107–139.

Hudak, P., Peyton Jones, S. L., Wadler, P., Boutel, B., Fairnbairn, J., Fasel, J. H., Guzmán, M. M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, R. B., Nikhil, R. S., Partain, W., and Peterson, J. 1992. Report on the programming language Haskell, a non-strict purely functional language. *ACM SIGPLAN Notices 27,* 5, R1–R164.

Hughes, J. 1989. Why functional programming matters. *The Computer Journal 32,* 2, 98–107.

Institute for New Generation Computer Technology, Ed. 1992. *Fifth Generation Computer Systems 1992.* Vol. 1,2. Ohmsha Ltd. and IOS Press. ISBN 4-274-07724-1.

Janson, S. 1994. AKL–a multiparadigm programming language. Ph.D. thesis, Uppsala University and SICS.

Janson, S. and Haridi, S. 1991. Programming paradigms of the Andorra Kernel Language. In *Logic Programming, Proceedings of the 1991 International Symposium (ISLP).* The MIT Press, San Diego, CA, USA, 167–183.

Janson, S., Montelius, J., and Haridi, S. 1993. Ports for objects in concurrent logic programs. In *Research Directions in Concurrent Object-Oriented Programming.* The MIT Press.

Kågedal, A., Van Roy, P., and Dumant, B. 1997. Logical State Threads 0.1. SICStus Prolog package available at `http://www.info.ucl.ac.be/people/PVR/implementation.html`.

Koller, A. and Niehren, J. 2000. Constraint programming in computational linguistics. In *Proceedings of the Eighth CSLI Workshop on Logic Language and Computation*, D. Barker-Plummer, D. Beaver, J. van Benthem, and P. S. di Luzio, Eds. CSLI Press.

Laburthe, F. and Caseau, Y. 1998. SALSA: A language for search algorithms. In *Proceedings of the Fourth International Conference on Principles and Practice of Constraint Programming (CP '98)*, M. Maher and J.-F. Puget, Eds. Lecture Notes in Computer Science, vol. 1520. Springer Verlag, Pisa, Italy, 310–324.

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

LEA, D. 2000. *Concurrent Programming in Java, Second Edition*. Addison-Wesley.

MAHER, M. 1987. Logic semantics for a class of committed-choice programs. In *Proceedings of the Fourth International Conference on Logic Programming (ICLP 87)*. The MIT Press, Melbourne, Australia, 858–876.

MAIER, D. AND WARREN, D. S. 1988. *Computing with logic: Logic Programming with Prolog*. Addison-Wesley.

MARRIOTT, K. AND STUCKEY, P. J. 1999. *Programming with Constraints: An Introduction*. The MIT Press.

MARTIN MÜLLER, TOBIAS MÜLLER, AND VAN ROY, P. 1995. Multiparadigm programming in Oz. In *Workshop on the Future of Logic Programming, International Logic Programming Symposium (ILPS 95)*, D. Smith, O. Ridoux, and P. Van Roy, Eds.

MCNAMARA, B. AND SMARAGDAKIS, Y. 2000. Functional programming in C++. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Montreal, Canada, 118–129.

MEHL, M. 1999. Doctoral dissertation. Ph.D. thesis, Universität des Saarlandes, Im Stadtwald, 66041 Saarbrücken, Germany.

MEHL, M., MÜLLER, T., SCHULTE, C., AND SCHEIDHAUER, R. 2000. Interfacing to C and C++. Tech. rep., Mozart Consortium. Jan. Available at `http://www.mozart-oz.org`.

MEHL, M., SCHEIDHAUER, R., AND SCHULTE, C. 1995. An abstract machine for Oz. In *Programming Languages, Implementations, Logics and Programs, Seventh International Symposium, PLILP'95*, M. Hermenegildo and S. D. Swierstra, Eds. Lecture Notes in Computer Science, vol. 982. Springer Verlag, Utrecht, The Netherlands, 151–168.

MEHL, M., SCHULTE, C., AND SMOLKA, G. 1998. Futures and by-need synchronization for Oz. Draft, Programming Systems Lab, Universität des Saarlandes. May.

MILNER, R., TOFTE, M., AND HARPER, R. 1990. *Definition of Standard ML*. The MIT Press, Cambridge, MA, USA.

MOSS, C. 1994. *Prolog++: The Power of Object-Oriented and Logic Programming*. Addison-Wesley.

MOZART CONSORTIUM. 2000. The Mozart programming system (Oz 3), version 1.1.0. Available at `http://www.mozart-oz.org`.

MÜLLER, T. 1999. Problem solving with finite set constraints in Oz. A tutorial. Tech. rep., Mozart Consortium. Jan. Available at `http://www.mozart-oz.org`.

MÜLLER, T. 2000. The Mozart constraint extensions tutorial. Tech. rep., Mozart Consortium. Jan. Available at `http://www.mozart-oz.org`.

NADATHUR, G. AND MILLER, D. 1995. *Higher-order logic programming*, Chapter 8. Volume 5 of Gabbay et al. Gabbay et al. (1995).

NIKHIL, R. S. 1994a. ID language reference manual version 90.1. Tech. Rep. Memo 284-2, MIT, Computation Structures Group. July.

NIKHIL, R. S. 1994b. An overview of the parallel language Id – a foundation for pH, a parallel dialect of Haskell. Tech. rep., Digital Equipment Corporation, Cambridge Research Laboratory.

NIKHIL, R. S. AND ARVIND. 2001. *Implicit Parallel Programming in pH*. Morgan Kaufmann.

ODERSKY, M. AND WADLER, P. 1997. Pizza into Java: translating theory into practice. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 97)*. Paris, France, 146–159.

PEYTON JONES, S. L. AND WADLER, P. 1993. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 71–84.

Rémy, D. and Vouillon, J. 1998. Objective ML: An effective object-oriented extension to ML. *Theory and Practive of Object Systems 4,* 1, 27–50.

Ritchie, D. M. 1987. Reflections on software research. See Ashenhurst and Graham (1987). 1983 Turing Award Lecture.

Robinson, J. A. 1965. A machine-oriented logic based on the resolution principle. *J. ACM 12*, 23–41.

Santos Costa, V., Warren, D. H. D., and Yang, R. 1991. Andorra-I: A parallel Prolog system that transparently exploits both And- and Or-parallelism. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. Williamsburg, VA, USA, 83–93.

Saraswat, V. and Rinard, M. 1990. Concurrent constraint programming. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL 90)*. San Francisco, CA, USA, 232–245.

Saraswat, V. A. 1993. *Concurrent Constraint Programming*. The MIT Press.

Scheidhauer, R. 1998. Doctoral dissertation. Ph.D. thesis, Universität des Saarlandes, Im Stadtwald, 66041 Saarbrücken, Germany. In German.

Schlichting, R. D. and Thomas, V. T. 1991. A multi-paradigm programming language for constructing fault-tolerant, distributed systems. Tech. Rep. TR 91-24, University of Arizona, Department of Computer Science. Oct.

Schulte, C. 1997a. Oz Explorer: A visual constraint programming tool. In *Proceedings of the Fourteenth International Conference on Logic Programming (ICLP 97)*, L. Naish, Ed. The MIT Press, Leuven, Belgium, 286–300.

Schulte, C. 1997b. Programming constraint inference engines. In *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming*, G. Smolka, Ed. Lecture Notes in Computer Science, vol. 1330. Springer Verlag, Schloß Hagenberg, Austria, 519–533.

Schulte, C. 1999a. Comparing trailing and copying for constraint programming. In *Proceedings of the 1999 International Conference on Logic Programming (ICLP 99)*, D. De Schreye, Ed. The MIT Press, Las Cruces, NM, USA, 275–289.

Schulte, C. 1999b. Oz Explorer–Visual constraint programming support. Tech. rep., Mozart Consortium. Jan. Available at `http://www.mozart-oz.org`.

Schulte, C. 2000a. Parallel search made simple. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, N. Beldiceanu, W. Harvey, M. Henz, F. Laburthe, E. Monfroy, T. Müller, L. Perron, and C. Schulte, Eds. Number TRA9/00. 55 Science Drive 2, Singapore 117599, 41–57.

Schulte, C. 2000b. Doctoral dissertation. Ph.D. thesis, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany.

Schulte, C. 2000c. Programming deep concurrent constraint combinators. In *Practical Aspects of Declarative Languages, Second International Workshop, PADL 2000*, E. Pontelli and V. S. Costa, Eds. Lecture Notes in Computer Science, vol. 1753. Springer Verlag, Boston, MA, USA, 215–229.

Schulte, C. 2002. *Programming Constraint Services*. Lecture Notes in Artificial Intelligence, vol. 2302. Springer Verlag, Berlin, Germany.

Schulte, C. and Smolka, G. 1994. Encapsulated search in higher-order concurrent constraint programming. In *Logic Programming: Proceedings of the 1994 International Symposium*, M. Bruynooghe, Ed. The MIT Press, Ithaca, NY, USA, 505–520.

Schulte, C. and Smolka, G. 1999. Finite domain constraint programming in Oz. A tutorial. Tech. rep., Mozart Consortium. Jan. Available at `http://www.mozart-oz.org`.

Schulte, C., Smolka, G., and Würtz, J. 1994. Encapsulated search and constraint programming in Oz. In *Second Workshop on Principles and Practice of Constraint Programming*, A. H. Borning, Ed. Lecture Notes in Computer Science, vol. 874. Springer Verlag, Orcas Island, WA, USA, 134–150.

Shapiro, E. 1983. A subset of Concurrent Prolog and its interpreter. Tech. Rep. TR-003, Institute for New Generation Computer Technology (ICOT), Cambridge, Mass. Jan.

Shapiro, E., Ed. 1987. *Concurrent Prolog: Collected Papers*. Vol. 1-2. The MIT Press, Cambridge, Mass.

Shapiro, E. 1989. The family of concurrent logic programming languages. *ACM Comput. Surv. 21,* 3 (Sept.), 413–510.

Smolka, G. 1995a. The definition of Kernel Oz. In *Constraints: Basics and Trends*. Lecture Notes in Computer Science, vol. 910. Springer Verlag, 251–292.

Smolka, G. 1995b. The Oz programming model. In *Computer Science Today*. Lecture Notes in Computer Science, vol. 1000. Springer Verlag, 324–343.

Smolka, G. 1996. Problem solving with constraints and programming. *ACM Computing Surveys 28,* 4es (Dec.). Electronic Section.

Smolka, G., Schulte, C., and Van Roy, P. 1995. PERDIO—Persistent and distributed programming in Oz. BMBF project proposal. Available at `http://www.ps.uni-sb.de`.

Smolka, G. and Treinen, R. 1994. Records for logic programming. *J. Log. Prog. 18,* 3 (Apr.), 229–258.

Sommerville, I. 1992. *Software Engineering*. Addison-Wesley.

Steele, Jr., G. L. 1984. *Common LISP: The Language*. Digital Press.

Sterling, L. and Shapiro, E. 1986. *The Art of Prolog–Advanced Programming Techniques*. Series in Logic Programming. The MIT Press.

Thompson, K. 1987. Reflections on trusting trust. See Ashenhurst and Graham (1987). 1983 Turing Award Lecture.

Thompson, S. 1999. *Haskell: The Craft of Functional Programming, Second Edition*. Addison-Wesley.

Tick, E. 1995. The deevolution of concurrent logic programming. *J. Log. Prog. 23,* 2 (May), 89–123.

Ueda, K. 1985. Guarded Horn Clauses. In *Logic Programming '85, Proceedings of the 4th Conference*, E. Wada, Ed. Lecture Notes in Computer Science, vol. 221. Springer Verlag, Tokyo, Japan, 168–179.

Van Hentenryck, P. 1999. *The OPL Programming Language*. The MIT Press. Software available from ILOG France.

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

Van Roy, P. 1989b. VLSI-BAM Diagnostic Generator. Prolog program to generate assembly language diagnostics, Aquarius Project, UC Berkeley, Alvin Despain.

Van Roy, P. 1999a. Logic programming in Oz with Mozart. In *Proceedings of the 1999 International Conference on Logic Programming (ICLP 99)*, D. De Schreye, Ed. The MIT Press, Las Cruces, NM, USA, 38–51. Mild version.

Van Roy, P. 1999b. 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.

Van Roy, P. and Despain, A. 1992. High-performance logic programming with the Aquarius Prolog compiler. *IEEE Computer*, 54–68.

Van Roy, P. and Haridi, S. 2002. *Concepts, Techniques, and Mod-*

els of Computer Programming. Book in progress. Draft available at `http://www.info.ucl.ac.be/people/PVR/book.html`.

Van Roy, P., Haridi, S., and Brand, P. 1999. Distributed programming in Mozart – A tutorial introduction. Tech. rep., Mozart Consortium. Jan. Available at `http://www.mozart-oz.org`.

Van Roy, P., Mehl, M., and Scheidhauer, R. 1996. Integrating efficient records into concurrent constraint programming. In *Proceedings of the Eighth International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP '96).* Springer Verlag, Aachen, Germany.

Wadler, P. 1992. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 92).* Albuquerque, NM, USA, 1–14. Invited talk.

Wikström, C. 1994. Distributed programming in Erlang. In *1st International Symposium on Parallel Symbolic Computation (PASCO 94).* World Scientific, Singapore, 412–421.