Encapsulated Search and Constraint Programming in Oz

Christian Schulte, Gert Smolka, and Jörg Würtz

German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, D–66123 Saarbrücken, Germany
{schulte,smolka,wuertz}@dfki.uni-sb.de

Abstract. Oz is an attempt to create a high-level concurrent programming language providing the problem solving capabilities of logic programming (i.e., constraints and search). Its computation model can be seen as a rather radical extension of the concurrent constraint model providing for higher-order programming, deep guards, state, and encapsulated search. This paper focuses on the most recent extension, a higher-order combinator providing for encapsulated search. The search combinator spawns a local computation space and resolves remaining choices by returning the alternatives as first-class citizens. The search combinator allows to program different search strategies, including depth-first, indeterministic one solution, demand-driven multiple solution, all solutions, and best solution (branch and bound) search. The paper also discusses the semantics of integer and finite domain constraints in a deep guard computation model.

1 Introduction

Oz [9, 2, 3, 7, 1] is an attempt to create a high-level concurrent programming language providing the problem solving capabilities of logic programming (i.e., constraints and search). Its computation model can be seen as a rather radical extension of the concurrent constraint model [6] providing for higher-order programming, deep guards, state, and encapsulated search. This paper focuses on the most recent extension, a higher-order combinator providing for encapsulated search. The search combinator spawns a local computation space and resolves remaining choices by returning the alternatives as first-class citizens. The search combinator allows to program different search strategies, including depth-first, indeterministic one solution, demand-driven multiple solution, all solutions, and best solution (branch and bound) search. The paper also discusses the semantics of integer and finite domain constraints in a deep guard computation model, which is an interesting issue since these constraints cannot be realized with their declarative semantics (due to intractability and even undecidability of satisfiability and entailment).

The idea behind our search combinator is simple and new. It exploits the fact that Oz is a higher-order language. The search combinator is given an expression E and a variable x (i.e., a predicate x/E) with the idea that E (which declaratively reads as a logic formula) is to be solved for x. The combinator spawns a

local computation space for E, which evolves until it fails or becomes stable (a property known from AKL). If the local computation space evolves to a stable expression $(A \vee B) \wedge C$, the two alternatives are returned as predicates:

$$x/(A \vee B) \wedge C \quad \rightarrow \quad x/A \wedge C,\ \ x/B \wedge C \ .$$

If the local computation space evolves to a stable expression C not containing a distributable disjunction, it is considered solved and the predicate x/C is returned.

We now relate Oz to AKL and cc(FD), two first-order concurrent constraint programming languages having important aspects in common with Oz.

AKL [4] is a deep guard language aiming like Oz at the integration of concurrent and logic programming. AKL can encapsulate search. AKL admits distribution of a nondeterminate choice in a local computation space spawned by the guard of a clause when the space has become stable (a crucial control condition we have also adopted in Oz). In AKL, search alternatives are not available as first-class citizens. All solutions search is provided through an extra primitive. Best solution and demand-driven multiple solution search are not expressible.

cc(FD) [10] is a constraint programming language specialized for finite domain constraints. It employs a Prolog-style search strategy and three concurrent constraint combinators called cardinality, constructive disjunction, and blocking implication. It is a compromise between a flat and a deep guard language in that combinators can be nested into combinators, but procedure calls (and hence nondeterminate choice) cannot. Encapsulated best solution search is provided as a primitive, but its control (e.g., stability) is left unspecified.

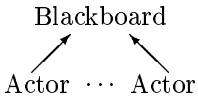
The paper is organized as follows. Section 2 gives an informal presentation of Oz's computation model, and Sect. 3 relates Oz to logic programming by means of examples. Section 4 shows how encapsulated and demand-driven search can be integrated into a reactive language. Section 5 presents the search combinator, and Sect. 6 shows how the search strategies mentioned above can be programmed with it. Section 7 discusses how integer and finite domain constraints are accommodated in Oz. Section 8 puts everything together by showing how the N-Queens problem can be solved in Oz.

2 Computation Spaces, Actors, and Blackboards

The computation model underlying Oz generalizes the concurrent constraint model (CC) [6] by providing for higher-order programming, deep guard combinators, and state. Deep guard combinators introduce local computation spaces, as in the concurrent constraint language AKL [4]. Recall that there is only one computation space in CC.

In [8] we give a formal model of computation in Oz, consisting of a calculus rewriting expressions modulo a structural congruence relation, similar to the set-up of the π-calculus [5]. For the purposes of this paper, an informal presentation of Oz's computation model, ignoring state, will suffice.

A computation space consists of a number of actors[1] connected to a blackboard.



The actors read the blackboard and reduce once the blackboard contains sufficient information. The information on the blackboard increases monotonically. When an actor reduces, it may put new information on the blackboard and create new actors. As long as an actor does not reduce, it does not have an outside effect. The actors of a computation space are short-lived: once they reduce they disappear. Actors may spawn local computation spaces.

The blackboard stores a constraint (constraints are closed under conjunction, hence one constraint suffices) and a number of named abstractions (to be explained later). Constraints are formulas of first-order predicate logic with equality that are interpreted in a fixed first-order structure called the Oz Universe. For the purposes of this paper it suffices to know that the Oz Universe provides rational trees (as in Prolog II) and integers. The constraint on the blackboard is always satisfiable in the Oz Universe. We say that a blackboard entails a constraint ψ if the implication $\phi \to \psi$ is valid in the Oz Universe, where ϕ is the constraint stored on the blackboard. We say that a blackboard is consistent with a constraint ψ if the conjunction $\phi \wedge \psi$ is satisfiable in the Oz Universe, where ϕ is the constraint stored on the blackboard. Since the constraint on the blackboard can only be observed through entailment and consistency testing, it suffices to represent it modulo logical equivalence.

There are several kinds of actors. This section will introduce elaborators, conditionals, and disjunctions.

An elaborator is an actor executing an expression. The expressions we will consider in this section are defined as follows:

$$
\begin{aligned}
E ::= {} & \phi \mid E_1\ E_2 \mid \textsf{local}\ x\ \textsf{in}\ E\ \textsf{end} \\
& \mid \textsf{proc}\ \{x\ y_1 \ldots y_n\}\ E\ \textsf{end} \mid \{x\ y_1 \ldots y_n\} \\
& \mid \textsf{if}\ C_1\ [\!]\ \ldots\ [\!]\ C_n\ \textsf{else}\ E\ \textsf{fi} \mid \textsf{or}\ C_1\ [\!]\ \ldots\ [\!]\ C_n\ \textsf{ro} \\
C ::= {} & E_1\ \textsf{then}\ E_2 \mid x_1 \ldots x_n\ \textsf{in}\ E_1\ \textsf{then}\ E_2
\end{aligned}
$$

Elaboration of a constraint ϕ checks whether ϕ is consistent with the blackboard. If this is the case, ϕ is conjoined to the constraint on the blackboard; otherwise, the computation space is marked failed and all its actors are cancelled. Elaboration of a constraint corresponds to the eventual tell operation of CC.

Elaboration of a concurrent composition $E_1\ E_2$ creates two separate elaborators for E_1 and E_2.

[1] Oz's actors are different from Hewitt's actors. We reserve the term agent for longer-lived computational activities enjoying persistent and first-class identity.

Elaboration of a variable declaration **local** x **in** E **end** creates a new variable (local to the computation space) and an elaborator for the expression E. Within the expression E the new variable is referred to by x. Every computation space maintains a finite set of local variables.

Elaboration of a procedure definition **proc** $\{x\ y_1 \ldots y_n\}$ E **end** chooses a fresh name a, writes the named abstraction $a{:}\, y_1 \ldots y_n/E$ on the blackboard, and creates an elaborator for the constraint $x = a$. Names are constants denoting pairwise distinct elements of the Oz Universe; there are infinitely many. Since abstractions are associated with fresh names when they are written on the blackboard, a name cannot refer to more than one abstraction.

Elaboration of a procedure application $\{x\ y_1\ \ldots\ y_n\}$ waits until the blackboard entails $x = a$ and contains a named abstraction $a{:}\, x_1 \ldots x_n/E$, for some name a. When this is the case, an elaborator for the expression $E[y_1/x_1 \ldots y_n/x_n]$ is created ($E[y_1/x_1 \ldots y_n/x_n]$ is obtained from E by replacing the formal arguments $x_1, \ldots, x_n$ with the actual arguments $y_1, \ldots, y_n$).

This simple treatment of procedures provides for all higher-order programming techniques. By making variables denote names rather than higher-order values, we obtain a smooth combination of first-order constraints with higher-order programming.

The elaboration of conditional expressions is more involved. We first consider the special case of a one clause conditional with flat guard.

Elaboration of **if** ϕ **then** E_1 **else** E_2 **fi** creates a conditional actor, which waits until the blackboard entails either ϕ or $\neg\phi$. If the blackboard entails ϕ [$\neg\phi$], the conditional actor reduces to an elaborator for E_1 [E_2]. In CC, such a conditional can be expressed as a parallel composition $(\mathbf{ask}\,\phi \rightarrow E_1)\,||\,(\mathbf{ask}\,\neg\phi \rightarrow E_2)$ of two ask clauses.

Elaboration of a conditional expression **if** C_1 $[]\ldots[]$ C_n **else** E **fi** creates a conditional actor spawning a local computation space for each clause C_i. A clause takes the form

$$x_1 \ldots x_k \textbf{ in } E \textbf{ then } D$$

where the local variables $x_1, \ldots, x_k$ range over both the guard E and the body D of the clause. We speak of a deep guard if E is not a constraint. In Oz, any expression can be used as a guard. This is similar to AKL and in contrast to CC, where guards are restricted to constraints. The local computation space for a clause

$$x \textbf{ in } E \textbf{ then } D$$

(clauses with no or several local variables are dealt with similarly) is created with an empty blackboard and an elaborator for the expression **local** x **in** E **end**.

Constraints from the global blackboard (the blackboard of the computation space the conditional actor belongs to) are automatically propagated to local spaces by elaborating them in the local spaces (propagation of global constraints can fail local spaces). Moreover, named abstractions from global blackboards are copied to local blackboards (conflicts cannot occur).

We say that a clause of a conditional actor is entailed if its associated computation space S is not failed, S has no actors left, and the global board entails

$\exists \overline{y}\, \phi$, where $\overline{y}$ are the local variables of S and ϕ is the constraint of the blackboard of S. Entailment of a local space is a stable property, (i.e., remains to hold when computation proceeds).

A conditional actor must wait until either one of its clauses is entailed or all its clauses (i.e., their associated local spaces) are failed.

If all clauses of a conditional actor **if** C_1 [] ... [] C_n **else** E **fi** are failed, the conditional actor reduces to an elaborator for the expression E (the else constituent of the conditional).

If a clause x_i **in** E_i **then** D_i of a conditional actor is entailed, the other clauses and their associated spaces are discarded, the space associated with the entailed clause is merged with the global space (conflicts cannot occur), and the conditional actor reduces to an elaborator for D_i (the body of the clause).

Elaboration of a disjunctive expression **or** C_1 [] ... [] C_n **ro** creates a disjunctive actor spawning a local computation space for every clause $C_1, \ldots, C_n$. The local spaces are created in the same way as for conditionals. As with conditional clauses, constraints and named abstractions from the global blackboard are automatically propagated to local blackboards.

A disjunctive actor must wait until all but possibly one of its clauses are failed, or until a clause whose body is the trivial constraint **true** is entailed. In the latter case, the disjunctive actor just disappears (justified by the equivalence $A \wedge (A \vee B) \equiv A$). If all clauses of a disjunctive actor are failed, the space of the disjunctive actor is failed (i.e., all its actors are cancelled). If all but one clause of a disjunctive actor are failed, it reduces with the unfailed clause. This is done in two steps. First, the space associated with the unfailed clause is merged with the global space, and then an elaborator for the body of the clause is created. The merge of the local with the global space may fail because the local constraint may be inconsistent with the global constraint. In this case the global space will be failed.

3 Example: Length of Lists

This section clarifies how Oz relates to logic programming and Prolog.

The Horn clauses

```
length(nil,0)
length(X|Xr, s(M)) ← length(Xr,M)
```

define a predicate length(Xs,N) that holds if Xs is a list of length N. Numbers are represented as trees 0, $s(0)$, $s(s(0))$, ..., and lists as trees $t_1|t_2|\ldots|t_n|$nil. The intended semantics of the clauses is captured by the equivalence

$$\begin{aligned}\mathsf{length}(\mathsf{Xs},\mathsf{N}) \leftrightarrow\ & \mathsf{Xs} = \mathsf{nil} \ \wedge\ \mathsf{N} = 0 \\ & \vee\ \exists \mathsf{X},\mathsf{Xr},\mathsf{M}\ (\mathsf{Xs} = \mathsf{X}|\mathsf{Xr} \wedge \mathsf{N} = \mathsf{s}(\mathsf{M}) \wedge \mathsf{length}(\mathsf{Xr},\mathsf{M}))\quad ,\end{aligned}$$

which is obtained from the Horn clauses by Clark's completion. The equivalence exhibits the relevant primitives and combinators of logic programming: constraints (i.e., Xs=nil), conjunction, existential quantification, disjunction, and

definition by equivalence. Given the equivalence, it is easy to define the length predicate in Oz:

```
proc {Length Xs N}
   or Xs=nil N=0 then true
   [] X Xr M in Xs=X|Xr N=s(M) then {Length Xr M}
   ro
end
```

There are two things that need explanation. First, the predicate is now referred to by a variable Length, as to be expected in a higher-order language. Second, the two disjunctive clauses have been divided into guards and bodies. The procedure application {Length Xr M} is put into the body to obtain a terminating operational semantics.

To illustrate the operational semantics of Length, assume that the procedure definition has been elaborated. Now we enter the expression

```
declare Xs N in {Length Xs N}
```

whose elaboration declares two new variables Xs and N and reduces the procedure application {Length Xs N} to a disjunctive actor. The **declare** expression is a variant of the **local** expression whose scope extends to expressions the programmer enters later. The disjunctive actor cannot reduce since there is no information about the variables Xs and N on the global blackboard. It now becomes clear why we did not write the recursive procedure application {Length Xr M} into the guard: this would have caused divergence.

Now we enter the constraint ('_' is a variable occurring only once)

```
N = s(s(_))
```

Since N = s(s(_)) is inconsistent with the constraint N=0 on the local blackboard, the first clause of the suspended disjunctive actor can now be failed and the disjunctive actor can reduce with its second clause. This will elaborate the recursive application {Length Xr M} and create a new disjunctive actor whose first clause fails immediately. This will create once more a new disjunctive actor, which this time cannot reduce. The global blackboard now entails Xs = _|_|_ and N = s(s(_)).

Next we enter the constraint

```
Xs = 1|2|nil
```

whose elaboration fails the second clause of the suspended disjunctive actor (since $x = \mathsf{nil}$ is inconsistent with $x = y|z$). Hence the suspended actor reduces with its first clause, no new disjunctive actor is created, and the blackboard finally entails Xs = 1|2|nil and N = s(s(0)).

The example illustrates important differences between Oz and Prolog: if there are alternatives (specified by the clauses of disjunctions or conditionals), Oz explores the guards of the alternatives concurrently. Only once it is safe to commit

to an alternative (e.g., because all other alternatives are failed or because the guard of a conditional clause is entailed), Oz will commit to it. In contrast, Prolog will eagerly commit to the first alternative if a choice is to be made, and backtrack if necessary.

A sublanguage of Oz enjoys a declarative semantics such that computation amounts to equivalence transformation [8]. The declarative semantics of a conditional

$$\textbf{if } x \textbf{ in } E_1 \textbf{ then } E_2 \textbf{ else } E_3 \textbf{ fi}$$

with only one clause is $\exists x(E_1 \wedge E_2) \vee (\neg\exists x E_1 \wedge E_3)$. Hence Oz can express negation $\neg E$ as **if** E **then false else true fi**.

The length predicate can also be defined in a functional manner using a conditional:

```
proc {Length Xs N}
   if Xs=nil then N=0
   [] X Xr M in Xs=X|Xr then N=s(M) {Length Xr M}
   else false fi
end
```

While the functional version has the same declarative reading as the disjunctive formulation, its operational semantics is different in that it will wait until information about its first argument is available. Thus

```
declare Xs N in N=s(s(0)) {Length Xs N}
```

will create a suspending conditional actor and not write anything on the global blackboard. On the other hand,

```
declare Xs N in Xs=_|_|nil {Length Xs N}
```

will write N=s(s(0)) on the global blackboard (although there is only partial information about Xs).

Oz supports functional syntax: the functional version of the length predicate can equivalently be written as:

```
fun {Length Xs}
   case Xs of nil then 0 [] X|Xr then s({Length Xr}) end
end
```

4 Encapsulated and Demand-driven Search

Given the length predicate of the previous section, Prolog allows to enumerate all pairs Xs, N such that length(Xs,N) is satisfied. This service can be obtained in Oz in a more flexible form. Oz provides search agents that can be given queries and be prompted for answers. These search agents take the form of objects, the basic concurrency abstraction of Oz.

An object is a procedure O taking a message M as argument. It encapsulates a reference to a data structure acting as the state of the object. A procedure application {O M} (the object is applied to the message) first competes for exclusive access to the object's state (necessary in a concurrent setting) and then applies the method requested by the message:

$$\text{method: state} \times \text{message} \rightarrow \text{state} \ .$$

This yields a new state which is released. The message indicates the method to be applied by a name that is mapped to the actual method by the object itself (so-called late binding).

Objects can be expressed in the computation model outlined in Sect. 2 if one further primitive, called cells, is added. Oz's higher-order programming facilities make it straightforward to obtain multiple inheritance of methods. For more information about objects in Oz we refer the reader to [2, 3, 1].

Now suppose Search is a search object as outlined above (any number of search objects can be created by inheritance from a predefined search object). First, we present it a query using the method query:

```
local Q in
   proc {Q A} local Xs N in A=Xs#N {Length Xs N} end end
   {Search query(Q)}
end
```

The query is specified by a unary predicate, so that solutions can be computed uniformly for one variable. Since we have existential quantification and pairing, this is no loss of generality. Using functional notation, we can write the above expression more conveniently as

```
{Search query(proc {A} local Xs N in A=Xs#N {Length Xs N} end end)}
```

Now we can request computation of the first solution by sending the message

```
{Search next}
```

which will produce the pair nil#0. Sending next (i.e., elaborating {Search next}) once more will produce (_|nil)#s(0), and so on. What happens when an solution is found can be specified by sending Search the message action(P), where P is a unary procedure to be applied to every solution found. The procedure P may, for instance, display solutions in a window or send them to other objects.

We remark that Prolog provides demand-driven search at the user interface, but not at the programming level. Aggregation in Prolog (i.e., bagof) is eager and will diverge if there are infinitely many solutions. In Oz, we can have any number of search objects at the same time and request solutions as required.

5 Solvers

We now introduce solvers, which are higher-order actors providing for encapsulated search. Many different search strategies can be programmed with solvers, ranging from demand-driven depth-first (as exemplified by the search object in the previous section) to best solution (branch and bound) strategies. In contrast to this rather informal introduction, in [7] one may find a calculus defining the presented ideas formally.

The key idea behind search in Oz is to exploit the distributivity law and proceed from $(A \vee B) \wedge C$ to $A \wedge C$ and $B \wedge C$. While Prolog commits to $A \wedge C$ first and considers $B \wedge C$ only upon backtracking, Oz makes both alternatives available as first-class citizens. To do this, the variable being solved for must be made explicit and abstracted from in the alternatives. For instance, if **or** $x = 1$ ▯ $x = 2$ **ro** is being solved for x, distribution will produce the abstractions **proc** $\{x\}$ $x = 1$ **end** and **proc** $\{x\}$ $x = 2$ **end**.

Solvers are created by elaboration of solve expressions

$$\textbf{solve}[x\colon E;\ u]$$

where x (the variable being solved for) is a local variable taking the expression E as scope. The variable u provides for output. The solver created by elaboration of the above expression spawns a local computation space for the expression

$$\textbf{local}\ x\ \textbf{in}\ E\ \textbf{end}$$

As with other local computation spaces, constraints and named abstractions are propagated from global blackboards to the local blackboards of solvers.

A solver can reduce if its local computation space is either failed or stable. A local computation space is called stable if it is blocked and remains blocked for every consistent extension of the global blackboard. A computation space is called blocked if it is not failed and none of its actors can reduce. Stability is known from AKL [4], where it is used to control nondeterministic promotion. Note that a local computation space is entailed if and only if it is stable and has no actor left.

If the local computation space of a solver has failed, the solver reduces to an elaborator for the constraint (u is the output variable)

$$u = \mathsf{failed}\ .$$

If the local computation space of a solver is stable and does not contain a disjunctive actor, the solver reduces to an elaborator for

$$u = \mathsf{solved}(\textbf{proc}\ \{x\}\ F\ \textbf{end})$$

where F is an expression representing the stable local computation space (the nested procedure definition has been explained in the previous section).[2] Abstracting the solution with respect to x is advantageous in case F does not fully

[2] The reader might be surprised by the fact that local computation spaces can be represented as expressions. This is however an obvious consequence of the fact that Oz's formal model [8] models computation states as expressions.

determine x; for instance, if F is **local** z **in** $x = f(z)$ **end**, different applications will enjoy different local variables z. A less general way to return the solution would be to reduce to an elaborator for $u = \mathsf{solved}(x)\ F$.

If the local computation space of a solver is stable and contains a disjunctive actor **or** C_1 [] ... [] C_n **ro**, the solver reduces to an elaborator for

$$u = \mathsf{distributed}(\textbf{proc}\ \{x\}\ \textbf{or}\ C_1\ \textbf{ro}\ F\ \textbf{end}\quad \textbf{proc}\ \{x\}\ \textbf{or}\ C_2\ []\ \ldots\ []\ C_n\ \textbf{ro}\ F\ \textbf{end})$$

where F is an expression representing the stable local computation space after deletion of the disjunctive actor. Requiring stability ensures that distribution is postponed until no other reductions are possible. This is important since repeated distribution may result in combinatorial explosion.

For combinatorial search problems it is often important to distribute the right disjunction and try the right clause first. Oz makes the following commitments about order: clauses are distributed according to their static order; solvers distribute the most recently created disjunctive actor; and elaboration proceeds from left to right, where suspended actors that become reducible are given priority (similar to Prologs with freeze). Taking the most recently created disjunctive actor for distribution seems to be more expressive than taking the least recently created one (see the first failure labeling procedure in Sect. 8).

Solvers cannot express breadth-first search if disjunctions with more than two clauses are used. This can be remedied by also returning the number of remaining clauses when a disjunctive actor is distributed.

Solve expressions are made available through a predefined procedure

```
proc {Solve P U} solve[X:{P X}; U] end
```

6 Search Strategies

We start with a function taking a query (i.e., a unary procedure) as argument and trying to solve it following a depth-first strategy:

```
fun {Depth Q}
   local S = {Solve Q} in
      case S of distributed(L R) then
         case {Depth L} of solved(_)=T then T else {Depth R} end
      else S end
   end
end
```

If no solution is found (but search terminates), failed is returned. If a solution is found, solved(A) is returned, where A is the abstracted solution. A procedure solving a query with Depth and displaying the result can be written as follows:

```
fun {One Q}
   local S = {Solve Q} in
      case S of distributed(L R) then
         if T in {One L}=solved(_)=T then T
         [] T in {One R}=solved(_)=T then T
         else failed fi
      else S end
   end
end
```

Fig. 1. Parallel one solution search.

```
proc {SolveAndBrowse Q}
   case {Depth Q} of failed then {Browse 'no solution found'}
   [] solved(A) then {Browse {A}}
   end
end
```

The search performed by Depth is sequential. Figure 1 shows an indeterministic search function One that explores alternatives in parallel guards.[3] The use of deep parallel guards provides a high potential for parallel execution.

Combinatorial optimization problems (e.g., scheduling) often require best solution search. Following a branch and bound strategy, this can be done as follows: once a solution is found, only solutions that are better with respect to a total order are searched for. With every better solution found, the constraints on further solutions can be strengthened, thus pruning the search space.

Figure 2 shows a function Best searching the best solution of a query Q with respect to a total order R (a binary procedure). The local function BAB takes two stacks Fs and Bs of alternatives and the best solution found so far as arguments (if no solution has been found so far, failed is taken as last argument) and returns the best solution. Alternatives which are already constrained to produce a better solution than S reside on the foreground stack Fs, and the remaining alternatives reside on the background stack Bs. If the foreground stack is empty, an alternative B from the background stack is taken. The query A obtained from constraining B to solutions better than S (the best solution so far) is expressed as follows:

```
A = proc {X} {R {S} X} {B X} end
```

If a new and better solution is obtained, all nodes from the foreground stack are moved to the background stack so that they will be correctly constrained before they are explored.

The program in Fig. 3 defines an object Search realizing the functionality described in Sect. 4. The object must be initialized with messages query(Q)

[3] This search function was suggested to us by Sverker Janson.

```
fun {Best Q R}
   local
      fun {BAB Fs Bs S}
         case Fs of nil then
            case Bs of nil then S
            [] B|Br then {BAB (proc {X} {R {S} X} {B X} end)|nil Br S}
            end
         [] F|Fr then
            case {Solve F} of failed then {BAB Fr Bs S}
            [] solved(T) then {BAB nil {Append Fr Bs} T}
            [] distributed(L R) then {BAB L|R|Fr Bs S}
            end
         end
      end
   in {BAB Q|nil nil failed} end
end
```

Fig. 2. Best solution search.

and action(A) fixing the query to be solved and the action to be taken when a solution is found, respectively. The attribute stack stores the unexplored alternatives. If a solution is requested with the method next, the alternatives on the stack are explored following a depth-first strategy. If no alternatives are left on the stack, the specified action is applied to the atom failed.

The search object illustrates object-oriented constraint programming in Oz. More sophisticated search strategies, for instance iterated depth-first search, can be obtained by refining Search using inheritance.

```
create Search from UrObject
   meth action(A)  action←A end
   meth query(Q)  stack←Q|nil end
   meth next
      case @stack of nil then {@action failed}
      [] N|Nr then
         case {Solve N} of failed then stack←Nr  ⟪next⟫
         [] solved(S) then stack←Nr  {@action solved(S)}
         [] distributed(L R) then stack←L|R|Nr  ⟪next⟫
         end
      end
   end
end
```

Fig. 3. Demand driven depth-first search.

7 Integers and Finite Domains

An implementation of the presented computation model must come with efficient and incremental algorithms for deciding satisfiability and entailment of constraints. This means that a programming language must drastically restrict the constraints a programmer can actually use. For instance, addition and multiplication of integers cannot be made available as purely declarative constraints since satisfiability of conjunctions of such constraints is undecidable (Hilbert's tenth problem).

The usual way to deal with this problem is to base the implementation on incomplete algorithms for satisfiability and entailment (e.g., delay nonlinear arithmetic constraints until they are linear). Consequently, constraints are not anymore fully characterized by their declarative semantics, and the programmer must understand their operational semantics.

In Oz, we make a distinction between basic and virtual constraints. Basic constraints are what has been called constraints so far. Their semantics is given purely declaratively by the Oz Universe. Oz is designed such that the programmer can only write basic constraints whose declarative semantics can be faithfully realized by the implementation (i.e., sound and complete algorithms for satisfiability and entailment). Virtual constraints are procedures whose operational semantics is sound but incomplete with respect to the declarative semantics of the corresponding logic constraint. A typical example of a virtual constraint is the length predicate for lists defined in Sect. 3.

Most constraints expressible over the Oz Universe are only available through predefined virtual constraints (i.e., with incomplete operational semantics). A typical example is addition of integers, whose definition is as follows:

```
proc {`+` X Y Z}
   if int(X) int(Y) isdet[X] isdet[Y] then plus(X,Y,Z) else false fi
end
```

Here plus(X,Y,Z) is the basic constraint expressing integer addition (partial functions are avoided by using relations), int(X) is the basic constraint expressing that X is an integer, and isdet[X] creates an actor that disappears as soon as there is a constant a in the signature of the Oz Universe such that X=a is entailed by the blackboard. Clearly, there is no difficulty in implementing the virtual constraint {`+` X Y Z}. Moreover, its semantics is fully defined in terms of the computation model outlined in Sect. 2 (extended with the isdet[X] actor, of course).

The virtual constraint

```
proc {IsInt X}
   if int(X) isdet[X] then true else false fi
end
```

will fail if the blackboard entails that X is no integer, and disappear (important for deep guards) if there is an integer n such that the blackboard entails X=n.

A further example is the predefined virtual constraint

```
proc {`≤` X Y}
   if {IsInt X} {IsInt Y} then Ie(X,Y) else false fi
end
```

where and Ie(X,Y) is the basic constraint expressing the canonical order on integers.

The predefined virtual constraint

```
proc {FdIn X L U}
   if {IsInt L} {IsInt U} then Ie(L,X) Ie(X,U) Ie(Inf,L) Ie(U,Sup) else false fi
end
```

makes it possible to constrain a variable X to a finite domain L..U (i.e., the value of X must be an integer between L and U). There variables Inf and Sup are predefined by the implementation and fix the maximal size of finite domains (i.e., there are only finitely many finite domains).

Another important predefined virtual constraint is

```
proc {FdNec X C}
   if {FdIn X Inf Sup} {IsInt C} then X ≠ C else false fi
end
```

whose declarative reading says that X is a finite domain variable different from C ($X \neq C$ is a basic constraint).

Figure 4 shows the definition of a virtual constraint $X \leq' Y$ enforcing domain consistency for finite domain variables (the infix operators $\leq$, $+$, and $-$ expand to applications of the corresponding virtual constraints). For instance, elaboration of the expression

```
local X Y in
   {FdIn X 3 7} {FdIn Y 7 24}
   if X ≤' Y then {Browse yes} else {Browse no} fi
end
```

will reduce the conditional actor to {Browse yes}, and elaboration of

```
{FdIn X 3 7} {FdIn Y 7 24} Y ≤' X
```

will constrain X and Y to 7.

With the outlined techniques we can formally define all finite domain constraints as virtual constraints such that a faithful and efficient implementation is possible. To our knowledge, this is the first formal semantics for finite domain constraints in a deep guard computation model.

To define heuristics such as first failure labeling (see next section), we need a reflective primitive. The actor

$$\textbf{reflect}[x;\ y]$$

```
proc {`≤′` X Y}
   if {FdIn X Inf Sup} {FdIn Y Inf Sup} then
     local
        proc {LE Xl Xu Yl Yu}
           if X=Y then true
           [] Xu≤Yl then true
           [] {FdIn X Xl+1 Sup} then {FdIn Y Xl+1 Sup} {LE Xl+1 Xu Yl Yu}
           [] {FdIn X Inf Xu−1} then {LE Xl Xu−1 Yl Yu}
           [] {FdIn Y Yl+1 Sup} then {LE Xl Xu Yl+1 Yu}
           [] {FdIn Y Inf Yu−1} then {FdIn X Inf Yu−1} {LE Xl Xu Yl Yu−1}
           fi
        end
     in {LE Inf Sup Inf Sup} end
   else false fi
end
```

Fig. 4. The virtual constraint $\mathsf{X} \leq' \mathsf{Y}$.

can reduce as soon as the blackboard constrains the variable x to a finite domain. It will then reduce to an elaborator for the constraint $y = n_1|\ldots|n_k|\mathsf{nil}$, where $n_1|\ldots|n_k|\mathsf{nil}$ is the shortest list in ascending order such that the blackboard entails the constraint $x = n_1 \vee \ldots \vee x = n_k$. Note that the reflection actor is different from all other actors in that its reduction may have different effect if it is postponed.

8 Example: N-queens

Figure 5 shows an Oz program solving the n-queens problem (place n queens on an $n \times n$ chessboard such that no queen is attacked by another queen). The predicate {Queens N Xs} is satisfied iff the list Xs represents a solution to the n-queens problem. The list Xs has length N, where every element is an integer between 1 and N. The ith element of Xs specifies in which row the queen in the ith column is placed. The solutions to the 100-queens problem, say, can be obtained by providing the search object of Sect. 6 with the query

```
{Search query(proc {Xs} {Queens 100 Xs} end)}
```

The procedure {Consistent Xs Ys} iterates through the columns of the board, where Ys are the columns already constrained and Xs are the columns still to be constrained. Since a queen only imposes its constraints once it is determined (i.e., {IsInt X} can reduce), there are at most N actors spawned before a distribution.

The procedure {Label Xs} labels the elements of Xs. Different labeling strategies are possible. Figure 6 shows a labeling procedure realizing the first-fail heuristic (label variables with fewest remaining values first). The procedure FdSize yields the number of values still possible for a finite domain variable,

```
local
   proc {NoAttack Xs Y I}
      case Xs of nil then true
      [] X|Xr then
         {FdNec X Y} {FdNec X Y+I} {FdNec X Y-I} {NoAttack Xr Y I+1}
      end
   end
   proc {Consistent Xs Ys}
      case Xs of nil then true
      [] X|Xr then
         if {IsInt X} then {NoAttack Xr X 1} {NoAttack Ys X 1} fi
         {Consistent Xr X|Ys}
      end
   end
   proc {Board I N Xs}
      if I=0 then Xs=nil
      else local X Xr in Xs=X|Xr {FdIn X 1 N} {Board I-1 N Xr} end fi
   end
in
   proc {Queens N Xs} {Board N N Xs} {Consistent Xs nil} {Label Xs} end
end
```

Fig. 5. The n-queens problem.

```
proc {Label Xs}
   case {Sort {Filter Xs proc {X} {FdSize X}>1 end}
              proc {X Y} {FdSize X}<{FdSize Y} end} of nil then true
   [] X|Xr then local M={FdMin X} in
                  or X=M then {Label Xr} [] {FdNec X M} then {Label X|Xr} ro
               end
   end
end
```

Fig. 6. First-failure labeling.

and FdMin yields the minimal value still possible. Both procedures can be expressed with the reflection actor of Sect. 7.

After all determined elements of Xs have been dropped with the higher-order procedure Filter, the remaining elements are sorted according to the current size of their domain. If X is the variable with the smallest domain, the disjunction or X=M then {Label Xr} [] {FdNec X M} then {Label X|Xr} ro is created, where M is the minimal possible value for X, and Xr are the remaining variables to be labeled.

Because of the use of the reflective procedures FdSize and FdMin, it is important that the labeling procedure is elaborated only after all constraints have been propagated. This is ensured by the fact that suspended actors are given priority once they become reducible, and that the application of Label appears last. Since

the most recently created disjunctive actor is distributed, the latter ensures that the disjunctive actor created by the labeling procedure is distributed even if there are further disjunctive actors (which is not the case in our example).

Acknowledgements

We thank Michael Mehl, Tobias Müller, Konstantin Popov, and Ralf Scheidhauer for discussions and implementing Oz. We also thank Sverker Janson for discussions of search issues.

The research reported in this paper has been supported by the Bundesminister für Forschung und Technologie (FTZ-ITW-9105), the Esprit Project ACCLAIM (PE 7195), and the Esprit Working Group CCL (EP 6028).

The Oz System and its documentation are available through anonymous ftp `ps-ftp.dfki.uni-sb.de` or through www `http://ps-www.dfki.uni-sb.de/`.

References

1. M. Henz, M. Mehl, M. Müller, T. Müller, J. Niehren, R. Scheidhauer, C. Schulte, G. Smolka, R. Treinen, and J. Würtz. The Oz Handbook. Research Report RR-94-09, DFKI, 1994. Available through anonymous ftp from `duck.dfki.uni-sb.de`.
2. M. Henz, G. Smolka, and J. Würtz. Oz—a programming language for multi-agent systems. In *13th International Joint Conference on Artificial Intelligence*, volume 1, pages 404–409, Chambéry, France, 1993. Morgan Kaufmann Publishers. Revised version will appear as [3].
3. M. Henz, G. Smolka, and J. Würtz. Object-oriented concurrent constraint programming in Oz. In P. van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming*. The MIT Press, 1994. To appear.
4. S. Janson and S. Haridi. Programming paradigms of the Andorra kernel language. In *Logic Programming, Proceedings of the 1991 International Symposium*, pages 167–186. The MIT Press, 1991.
5. R. Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.
6. V. A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages*, pages 232–245, Jan. 1990.
7. C. Schulte and G. Smolka. Encapsulated search in higher-order concurrent constraint programming. In *Logic Programming: Proceedings of the 1994 International Symposium*, Ithaca, New York, USA, Nov. 1994. MIT-Press. To appear.
8. G. Smolka. A calculus for higher-order concurrent constraint programming with deep guards. Research Report RR-94-03, DFKI, Feb. 1994.
9. G. Smolka. A foundation for higher-order concurrent constraint programming. Research Report RR-94-16, DFKI, June 1994. Also in *1st International Conference on Constraints in Computational Logics*, München, Germany, 7–9 September 1994.
10. P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation and evaluation of the constraint language cc(FD). Report CS-93-02, Brown University, Jan. 1993.

This article was processed using the LaTeX macro package with LLNCS style