Encapsulated Search for Higher-order Concurrent Constraint Programming[1]

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

Abstract

The paper presents an extension of the concurrent constraint model providing for higher-order programming, deep guards, and encapsulated search. The paper focuses on 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 described computation model is realized in Oz, a programming language and system under development at DFKI.

Keywords Concurrent constraint programming, higher-order programming, encapsulated search, search strategies, Oz.

1 Introduction

Oz [2, 3, 9, 8, 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 idea behind our search combinator is simple and new. It exploits

[1]Appears in: *Logic Programming: Proceedings of the 1994 International Symposium*, pages 505–520, edited by Maurice Bruynooghe, 13–17 November, 1994, Ithaca, New York, USA. The MIT-Press.

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) [11] 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.

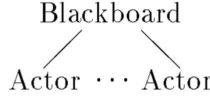
The paper is organized as follows. Section 2 gives an informal presentation of Oz's basic computation model (excluding search and state), and Section 3 relates it to logic programming by means of examples. Section 4 presents the search combinator informally, and Section 5 shows how the search strategies mentioned above can be programmed with it. Section 6 explains how to program heuristic labeling strategies, whereas Section 7 gives a formal operational semantics of the search combinator.

2 Computation Spaces, Actors, and Blackboards

This section gives an informal presentation of the basic computation model underlying Oz (see [8] for a formal presentation). It generalizes the concurrent constraint model (CC) [6] by providing for higher-order programming

and deep guard combinators. 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.

A computation space consists of a number of actors[2] 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 universe. For the purposes of this paper it suffices to know that the universe provides rational trees as in Prolog II. The constraint on the blackboard is always satisfiable in the universe. We say that a blackboard entails a constraint ψ if the implication $\phi \rightarrow \psi$ is valid in the 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 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:

$$\begin{array}{lcl} E & ::= & \phi \quad | \quad E_1\ E_2 \quad | \quad \textbf{local}\ x\ \textbf{in}\ E\ \textbf{end} \\ & | & \textbf{proc}\ \{x\ y_1 \ldots y_n\}\ E\ \textbf{end} \quad | \quad \{x\ y_1 \ldots y_n\} \\ & | & \textbf{if}\ C_1\ \square\ \ldots\ \square\ C_n\ \textbf{else}\ E\ \textbf{fi} \quad | \quad \textbf{or}\ C_1\ \square\ \ldots\ \square\ C_n\ \textbf{ro} \\ C & ::= & E_1\ \textbf{then}\ E_2 \quad | \quad x_1 \ldots x_n\ \textbf{in}\ E_1\ \textbf{then}\ E_2 \end{array}$$

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.

[2] 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 concurrent composition $E_1\ E_2$ creates two separate elaborators for E_1 and E_2.

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 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 a conditional expression **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 $(\mathbf{ask}\, \phi \rightarrow E_1) \mid\mid (\mathbf{ask}\, \neg\phi \rightarrow E_2)$ of two ask clauses.

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

$x_1 \ldots x_k$ **in** E **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 **in** E **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

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

which is obtained from the Horn clauses by Clark's completion. The equivalence exhibits the basic 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 = _|_|_    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 = 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    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 one 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]. For instance, the declarative semantics of a conditional **if** x **in** E_1 **then** E_2 **else** E_3 **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; for instance, 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 Solvers

We now introduce solvers, which are higher-order actors providing for encapsulated search.

The key idea behind solvers 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, yielding the transition

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

For instance, if **or** $x = 1 \;\square\; x = 2$ **ro** is being solved for x, distribution will produce the abstractions $x/x = 1$ and $x/x = 2$.

There is no loss in generality in solving for one variable. For instance, if we want to solve E for x and y, we can solve **local** $x\ y$ **in** $z =$ pair$(x\ y)$ E **end** for z instead.

Solvers are created by elaboration of solve expressions

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

local x **in** E **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)

$u =$ 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 (the nested procedure definition has been explained in the previous section)

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

where F is an expression representing the stable local computation space.[3] Abstracting the solution with respect to x is advantageous in case F does

[3] 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 models computation states as expressions (see Section 7).

not fully determine x; for instance, if F is **local** z **in** $x = f(z)$ **end**, different applications of the abstracted solution will enjoy different local variables z. A less general way to return the solution would be to reduce to an elaborator for **local** x **in** $u = \mathsf{solved}(x)$ F **end**.

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

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

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 reduction proceeds from left to right, where not yet reducible actors are moved to newly created threads [9]. Taking the most recently created disjunctive actor for distribution seems to be more expressive than taking the least recently created one.

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.

Solvers are made available through a predefined procedure

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

taking the query to be solved (i.e., the pair x/E) as a unary procedure Q.

5 Search Strategies

Figure 1 shows a function taking a query as argument and trying to solve it following a depth-first strategy. 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:

```
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 2 shows an indeterministic search function One that explores alternatives in parallel guards.[4]

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

```
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
```

Figure 1: Depth-first search.

```
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
```

Figure 2: Parallel one solution search.

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 given total order are searched for. With every better solution found, the constraints on further solutions are strengthened, thus pruning the search space.

Figure 3 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 better solutions than S reside on the foreground stack Fs, and the other 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.

Figure 4 shows a procedure Demand realizing demand-driven depth-first search. The application {Demand Xs Q} installs an agent computing solutions

```
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
```

Figure 3: Best solution search.

```
local
   proc {Next Xs Qs}
      case Xs of nil then true
      [] X|Xr then
         case Qs of nil then X=failed Xr=nil
         [] Q|Qr then
            case {Solve Q} of failed then {Next Xs Qr}
            [] solved(S) then X=solved(S) {Next Xr Qr}
            [] distributed(L R) then {Next Xs L|R|Qr}
            end
         end
      end
   end
in proc {Demand Xs Q} {Next Xs Q|nil} end end
```

Figure 4: Demand-driven depth-first search.

of the query Q as requested by the stream Xs. Constraining Xs to X|Xr requests the first solution. If a solution is found, X is constrained to solved(S), where S is the abstracted solution, and further solutions can be requested by constraining Xr. If no solution is found, X is constrained to failed and Xr to nil. If no further solutions are desired, the search agent can be terminated by closing its stream (i.e., constraining it to nil).

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 concurrent search agents and request solutions as required.

6 Heuristic Labeling Strategies

Finite domain problems are typically solved with heuristic labeling strategies (e.g., first failure) [10]. Such strategies first propagate all constraints and then create a choice point for a variable, which is selected by a heuristic taking the current domains of some finite domain variables as input. It is crucial that the domains of variables are inspected only after all constraints have been propagated.

This control is straightforward to achieve in a sequential language like Prolog (by eager waking). Surprisingly, it is also easily expressed in our concurrent setting. The idea is to wrap the labeling into a disjunction

```
or true then  inspect domains
              select a variable
              create a choice point
[] true then false
ro
```

The disjunction is only distributed once the local computation space of the solver is stable, which means that all global information (possibly computed by concurrent agents) has arrived and that all local constraint propagation is done.

7 Formal Semantics

This section gives a formal operational semantics of the solve combinator. This is done by extending the Calculus B defined in [8]. We assume familiarity with Calculus B.

The computation model is defined as a calculus consisting of an abstract syntax defining a class of expressions, a congruence relation on expressions, called structural congruence, and a reduction relation defined on the expressions modulo structural congruence. This setup is known from the π-calculus [5]. The calculus generalizes the informal computation model presented in the previous sections in that it leaves unspecified the order in which actors are reduced and disjunctions are distributed, and in that it is parameterized with respect to a general notion of constraint system.

Ignoring the order of reduction steps buys significant simplifications: the difference between expressions (static) and actors (dynamic) as well as the notion of elaboration can be dropped, and blackboards need not be represented explicitly.

7.1 Constraint Systems

Our notion of constraint system is based on first-order predicate logic with equality. A *constraint system* consists of a signature Σ (a set of constant, function and predicate symbols), a consistent theory Δ (a set of sentences over Σ having a model), and an infinite set of constants in Σ called *names*

x, y, z	:		*variable*	
a, b, c	:		*name*	
u, v, w	$::=$	$x \mid a$	*reference*	
ϕ	:		*constraint*	
E	$::=$	ϕ	*constraint*	
	$\mid$	$E_1 \wedge E_2$	*composition*	
	$\mid$	$\exists u\ E$	*declaration*	
	$\mid$	$a{:}\,\overline{x}/E$	*abstraction*	*($\overline{x}$ linear)*
	$\mid$	$u\overline{v}$	*application*	
	$\mid$	**if** D **else** E	*conditional*	
	$\mid$	**or** (D)	*disjunction*	
	$\mid$	**solve**$(x{:}\ E; uvw)$	*solver*	
C	$::=$	E_1 **then** $E_2 \mid \exists u\ C$	*clause*	
D	$::=$	$C \mid \bot \mid D_1 \vee D_2$	*collection*	

Figure 5: Abstract syntax.

satisfying two conditions: (1) for every two distinct names a, b: $\Delta \models \neg(a \doteq b)$; (2) for every two sentences ϕ, ψ over Σ such that ψ can be obtained from ϕ by permutation of names: $\Delta \models \phi \leftrightarrow \psi$.

Given a constraint system, we will call every formula over its signature a *constraint*. We use $\bot$ for the constraint that is always false, and $\top$ for the constraint that is always true. We will use the following equivalence relation for constraints: $\phi \models\!\mid_\Delta \psi \;:\Longleftrightarrow\; \forall\,(\phi \leftrightarrow \psi)$ is valid in every model of Δ.

7.2 Syntax

The abstract syntax of our calculus appears in Figure 5. It supposes that some constraint system is given, fixing infinite sets of variables, names and constraints. Variables and names are jointly referred to as *references*.

We use $\overline{u}$ to denote a possibly empty sequence of references. A sequence $\overline{u}$ is called *linear* if its elements are pairwise distinct.

Composition $E_1 \wedge E_2$ and variable declaration $\exists x\ E$ correspond to parallel composition and hiding in the π-calculus [5] and in CC [6], respectively.

An expression $a{:}\,\overline{x}/E$ represents a binding of the name a to the abstraction $\overline{x}/E$. For convenience, we call the entire expression $a{:}\,\overline{x}/E$ an "abstraction".

The syntactic category D represents multisets of clauses, where $\bot$ stands for the empty multiset and $\vee$ for multiset union.

We identify a conjunction $\phi_1 \wedge \phi_2$ of two constraints with the corresponding composition of constraints, and an existential quantification $\exists x\, \phi$ of a constraint ϕ with the corresponding declaration.

A solver $\mathbf{solve}(x\colon E; uvw)$ carries references u, v, w to three continuations (i.e., abstractions), which are applied when the local computation space E is failed, distributed, or solved. This formulation of the solve combinator avoids any further assumptions about the underlying constraint system. It can express the solve combinator of Section 4 if the constraint system provides for constructor trees.

Our calculus has the following constructs for binding references: A declaration $\exists u\, E$ binds u (a variable or a name) with scope E; an abstraction $a\colon\overline{x}/E$ binds its formal arguments $\overline{x}$ with scope E; a clausal declaration $\exists u\, C$ binds u (a variable or a name) with scope C; a solver $\mathbf{solve}(x\colon E; uvw)$ binds x with scope E; and quantification in constraints binds variables as in predicate logic. The *free variables* and *free names* of an expression are defined accordingly.

A procedure definition **proc** $\{x\, y_1 \ldots y_n\}$ E **end** (concrete syntax) translates into $\exists a\, (x = a \wedge a\colon y_1 \ldots y_n/E)$, and a procedure application $\{x\, y_1 \ldots y_n\}$ translates into $xy_1 \ldots y_n$. Note that declaration of names $\exists a\, E$ models creation of fresh names (see [8] for a discussion of this issue).

7.3 Structural Congruence

The *structural congruence* "$E_1 \equiv E_2$" of our calculus is a congruence on the set of expressions satisfying the following congruence laws (there are additional laws for disjunctions and conditionals not given here for lack of space; see [8]):

1. $E_1 \equiv E_2$ if E_1 and E_2 are equal up to renaming of bound references
2. $\wedge$ is associative, commutative and satisfies $E \wedge \top \equiv E$
3. $\exists u\, \exists v\; E \;\equiv\; \exists v\, \exists u\; E$
4. $\exists u\; E_1 \wedge E_2 \;\equiv\; \exists u\; (E_1 \wedge E_2)$ if u does not occur free in E_2
5. $\phi_1 \;\equiv\; \phi_2$ if $\phi_1 \models\!\!\dashv_\Delta \phi_2$
6. $x \doteq u \wedge E \;\equiv\; x \doteq u \wedge E[u/x]$ if u is free for x in E ($E[u/x]$ is obtained from E by replacing every free occurrence of x with u)
7. $\pi \wedge \mathbf{solve}(x\colon E; uvw) \;\equiv\; \pi \wedge \mathbf{solve}(x\colon \pi \wedge E; uvw)$ if π is a constraint or an abstraction such that x does not occur free in π.

Law (7) realizes propagation of constraints and abstractions from global to local blackboards.

7.4 Reduction

The reduction relation of our calculus is defined as a binary relation $E_1 \rightarrow E_2$ on the set of expressions satisfying the following reduction laws (there are

additional laws for disjunctions and conditionals not given here for lack of space; see [8]):

1. if $E_1 \equiv E_2$, $E_2 \rightarrow E_2'$, and $E_2' \equiv E_3$, then $E_1 \rightarrow E_3$
2. if $E_1 \rightarrow E_1'$, then $E_1 \wedge E_2 \rightarrow E_1' \wedge E_2$
3. if $E \rightarrow E'$, then $\exists u\, E \rightarrow \exists u\, E'$
4. if $E \rightarrow E'$, then $\mathbf{solve}(x\colon E; uvw) \rightarrow \mathbf{solve}(x\colon E'; uvw)$
5. $a\overline{u} \wedge a\colon \overline{x}/E \;\rightarrow\; E[\overline{u}/\overline{x}] \;\wedge\; a\colon \overline{x}/E$ if $\overline{x}$ and $\overline{u}$ are of equal length and $\overline{u}$ is free for $\overline{x}$ in E
6. $\mathbf{solve}(x\colon \bot \wedge E; uvw) \;\rightarrow\; u$
7. $\mathbf{solve}(x\colon E; uvw) \;\rightarrow\; \exists a\, \exists b\, (vab \wedge a\colon x/E_2 \wedge b\colon x/E_3)$ if $\exists x\, E$ stable, $E = \exists \overline{u}\; (E_1 \wedge \mathbf{or}\, (C \vee D))$, $E_2 = \exists \overline{u}\; (E_1 \wedge \mathbf{or}\, (C))$, and $E_3 = \exists \overline{u}\; (E_1 \wedge \mathbf{or}\, (D))$
8. $\mathbf{solve}(x\colon E; uvw) \;\rightarrow\; \exists a\, (wa \wedge a\colon x/E)$ if $\exists x\, E$ stable and not distributable.

An expression E is *failed* if there exists E' such that $E \equiv \bot \wedge E'$. An expression E is *stable* if, for every abstraction or satisfiable constraint π, the expression $\pi \wedge E$ is neither reducible nor failed. An expression E is called *distributable* if there exist $\overline{u}$, E' and D such that $E \equiv \exists \overline{u}\; (E' \wedge \mathbf{or}\, (D))$.

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).

Remark

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

[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, G. Smolka, and J. Würtz. Encapsulated search and constraint programming in Oz. In *Second Workshop on Principles and Practice of Constraint Programming*, Orcas Island, Washington, USA, May 1994. Springer-Verlag. 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. In J.-P. Jouannaud, editor, *1st International Conference on Constraints in Computational Logics*, Lecture Notes in Computer Science, München, Germany, 7–9 Sept. 1994. Springer-Verlag. Invited Lecture. To appear.

[10] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Programming Logic Series. The MIT Press, Cambridge, MA, 1989.

[11] 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.