Fondements de la Programmation: Concurrent Constraint Programming in OZ

Giovanni Coray - Vincenzo Pallotta

June 25, 2000

References:

- "The OZ programming model" Gert Smolka, Computer Science Today, LNCS 1000, 1995.
- "The Mozart programming system" `http://www.mozart-oz.org/`.

From CLP to Concurrent Constraint Programming

- In CLP we are committed with the operational semantics to follow a fixed type of computational schema (Resolution) and with a fixed number of built-in constraint theories.
- CHR generalizes CLP allowing users to define new constraint solvers.
- We want a more general computational model which combines features of different programming paradigms:
  - Constraint programming
  - Logic programming (Logic variables, non-determinism, search)
  - Higher-order Functional programming
  - Imperative programming
  - Object-oriented programming
  - Concurrent programming
  - Distributed programming

OZ Programming Model

- OZ is an extension of basic concurrent constraint model (CC) [see Saraswat93].
- It adds first-class procedures and stateful data structures.
- It subsumes higher-order functional programming and object-oriented programming.
- It allows the implementation of CLP-like problem solvers power by means of concurrent agents with encapsulated search.
- Application range over:
  - simulations
  - multi-agent systems
  - natural language processing
  - virtual reality
  - graphical user interfaces
  - optimization (scheduling, time-tabling, etc.)
  - configuration

The MOZART system

- It is an implementation of OZ which is claimed to be:
  - efficient
  - robust
  - interactive
- It is currently implemented on sequential hardware and supports distributed computation.
- Being a concurrent language there are in principle no inherent obstacles to a parallel implementation of OZ.
- It provides an interesting programming environment:
  - GNU Emacs programming interface
  - concurrent browser
  - object-oriented interface to TCL/TK for building GUI
  - interoperability features
  - incremental compilation
  - run-time system.

Computation Spaces

- A *Computation space* hosts a number of *tasks* connected to a *shared store*.
- Computations advance by *reduction* of tasks.
- Reduction of task is atomic and there is no parallelism at the abstraction level of OZ (i.e. parallelism is an implementation issue).
- Reduction of a task:
  - may manipulate the store
  - may create new tasks
  - erase the reduced task
- Task can *synchronize* on the store:
  - they become reducible only once the store satisfy certain conditions
  - reducibility does not depend on the task reduction strategy (monotonicity of synchronization).
- A *reduction strateg*y is needed to obtain:
  - fairness (several groups of task can advance simultaneously)
  - reactivity (time-bounded synchronization with outside events)
  - efficiency.

Constraint Store

- a special compartment of the shared store in the computation space
- it contains logical formulas over a constraint language closed under conjunction
- it can be considered as a logical conjunction of primitive constraints
- it contains (possibly partial) information about values of variables in the computation space
- it can be monotonically updated only by means of *tell* operations
  - the order of *tell* operations is insignificant (conjunction is associative and commutative).
  - it is kept always consistent (disallowing the insertion of constraint resulting into an inconsistent constraint store)
  - it can be represented modulo logic equivalence

Constraint Structures

- We consider first-order constraint structures $\chi = \langle \Sigma, \mathcal{C}, \mathcal{L}, \mathcal{T} \rangle$ together with the notion of constraint entailment:
- A constraint C *entails* a constraint D if $\mathcal{C} \models C \Rightarrow D$.
- A number of basic *built-in constraint structures* are provided by OZ:
  - Integer numbers
  - names
  - pairs (lists) over names and integers.
- *Primitive constraints* are:
  - $x = n$ (the value of the variable x is the number n)
  - $x = \xi$ (the value of the variable x is the name ξ)
  - $x = y|z$ (the value of the variable x is the pair having the value of the variable y as left and z as right component)
  - $x = y$ (the variable x and y have the same value).

Example
Let be $C = \{x = y, y = z|u, z = 3\}$ be the current constraint store. Then:

- $C \Rightarrow x = 3|u$
- $C \Rightarrow \neg(x = 3)$
- $C \not\Rightarrow y = 3|5$

Synchronization as Logic Entailment

- *Synchronization* of task is fully determined by its guard (i.e. a constraint) w.r.t. the constraint store:
  - a synchronized task becomes reducible when its guard become *entailed* by the current constraint store
  - logic monotonicity of the constraint store ensures task reduction stability.

OZ kernel language

Abstract Syntax for tasks

$$\begin{array}{rcll} E & ::= & C & \textit{constraint} \\ & | & E_1 \wedge E_2 & \textit{composition} \\ & | & \textbf{if}\, C\, \textbf{then}\, E_1\, \textbf{else}\, E_2 & \textit{conditional} \\ & | & \textbf{local}\, x\, \textbf{in}\, E\, \textbf{end} & \textit{declaration} \end{array}$$

- A computation space consists of tasks expressions E as defined above.

- The only synchronized tasks are conditionals. They synchronize becoming reducibles only once their guard is entailed by the constraint store.

Reduction of tasks

- The constraint task C *tells* the constraint C to the current the constraint store S provided that $S \wedge C$ is satisfiable. The resulting constraint store is a new constraint store logical equivalent to $S \wedge C$. In the case of an inconsistent resulting store, a *failure* is announced without changing the constraint store.

- The reduction of a composition task $E_1 \wedge E_2$ will create two new tasks E_1, E_2 which are added to the computation space.

- The reduction of a conditional task $\textbf{if}\, C\, \textbf{then}\, E_1\, \textbf{else}\, E_2$ in the store S creates:
  - E_1 if $S \Rightarrow C$ or
  - E_2 if $S \Rightarrow \neg C$.

- The reduction of $\textbf{local}\, x\, \textbf{in}\, E\, \textbf{end}$ in the store S will create a *fresh* variable y and the new task $E[y/x]$. Observe that if the computation space contains two identical declaration tasks $\{\textbf{local}\, x\, \textbf{in}\, E\, \textbf{end}, \textbf{local}\, x\, \textbf{in}\, E\, \textbf{end}\}$ then they will reduce to two different tasks $E[y/x]$ and $E[z/x]$ where y, z are distinct fresh variable.

Synchronized decomposition

- Telling constraints are used to assert information about the values of logical variables.

- It is possible to combine conditional and telling, in order to equate variable to components of non-primitive values (e.g. pairs). For instance the task

$$\textbf{if}\, \exists y \exists z (x = y|z)\, \textbf{then}\, x = u|v\, \textbf{else}\, E$$

will reduce to the constraint which equates the variable u and v to the components of x provided that x is a pair, and reduce to task E otherwise. More generally:

$$\textbf{if}\, x_1, \ldots, x_n\, \textbf{in}\, C\, \textbf{then}\, E_1\, \textbf{else}\, E_2$$

is an abbreviation for

$$\textbf{if}\, \exists x_1, \ldots, \exists x_n C\, \textbf{then}\, (\textbf{local}\, x_1\, \textbf{in} \cdots \textbf{local}\, x_n\, \textbf{in}\, (C \wedge E_1) \cdots \textbf{end}\, \textbf{else}\, E_2$$

which allows to rewrite the above example task in a more readable form as

$$\textbf{if}\, y, z\, \textbf{in}\, x = y|z\, \textbf{then}\, (u = y \wedge v = z)\, \textbf{else}\, E.$$

- Given a constraint store S, a variable x is bound to:
  - an integer n if $S \Rightarrow x = n$
  - a name ξ if $S \Rightarrow x = \xi$
  - a pair if $S \Rightarrow \exists y \exists z (x = y|z)$

First-class Procedures (1)

- A programming language provides first-class procedures if
  - procedures can create new procedures
  - procedures can have lexically scoped global variables
  - procedures are referred to by first-class values.

- An OZ procedure is a triple:

$$\xi : \overline{z}/E$$

where ξ is a name, $\overline{z}$ is a sequence of variables (formal parameters) and E is a task (the body).

- Procedure are contained in procedure store compartment of the shared store. There can be at most one procedure for one name.

- Procedure cannot be retracted from the procedure store.

- Procedure corresponds to well-known *closures*.

First-class Procedures (2)

- The language of OZ tasks is extended for procedures as follows:

$$\begin{array}{rcll} E & ::= & \textbf{proc}\, \{x\, z\}\, E\, \textbf{end} & \textit{definition} \\ & | & \{x\, y\} & \textit{application} \end{array}$$

- Reductions for $\textbf{proc}\, \{x\, z\}\, E\, \textbf{end}$ chooses a fresh name ξ and adds:

  $x = \xi$ to the constraint store

  $\xi : z/E$ to the procedure store

- Reduction for $\{x\, y\}$ waits until the procedure store contains $\xi : z/E$ such that $S \Rightarrow x = \xi$ and it reduces to the task E' which is obtained from E replacing all the free occurrences of z with y.

Higher-order concurrent functions in OZ

The function:

$$MkMap[V] : (V \to V) \to (List[V] \to List[V])$$

can be expressed as binary procedure:

```
proc {MkMap F Map}
  proc {Map Xs Ys}
    if X,Xr in Xs=X|Xr then
       local Y,Yr in Ys=Y|Yr {F X Y} {Map Xr Yr} end
    else Ys=nil fi
  end
end
```

where the concrete syntax of OZ is used (i.e. $E_1 \wedge E_2$ corresponds to E1 E2, a list $v_1, \ldots, v_n$ is represented as a nested pair $v_1|(...(v_n|nil))...)$, where nil is a name representing the empty list.
The procedure call:

```
{{MkMap F} 1|2|nil X}
```

is an abbreviation for:

```
local Map One Two A B in
   {MkMap F Map} One=1 Two=2 A=One|B B=Two|nil {Map A X}
end
```


Stateful computations

- OZ accounts for stateful computations by means of *cells* which are stored in cell store compartment of the shared store.
- A cell is a mutable binding of a name to a variable.
- Cells are provided by atomic read and write operations, making possible mutual exclusion and indeterminism as needed for many-to-one communication.
- Cells are useful to create *objects* and *concurrent data structures* which are used to as a communication medium between concurrent *agents*.

Cells

- The cell store contains a finite number of cells of the form:

  $$\xi : x$$

  where ξ is a name and x is a variable hosted in the cell store.
- There are two basic operations on cells:
  - `{NewCell X Y}`: chooses a fresh name ξ, tells the constraint $Y = \xi$, and writes the new cell $\xi : X$ into the cell store. Cells cannot be retracted.
  - `{Exchange X Y Z}`: waits until the cell store contains a cell $\xi : u$ such that the $S \Rightarrow X = \xi$. Then the task is reduced by updating the cell store with $\xi : Z$ and telling the constraint $Y = u$.
- Cells introduce *indeterminism*: the order of reductions for multiple exchanges on the same cell is unspecified.
- Cells are different than assignable variable in multi-threaded imperative language since:
  - OZ ensures mutual exclusion for concurrent exchanges (task reduction is atomic)
  - exchange task combines reading and writing into a single atomic operation.
- Cells may contains variables without values. Consumer of variables are automatically synchronized on the event that the value of the variable becomes known.
- Complex concurrent abstract data structure with state can be expressed by means of cells and encapsulated within a procedural abstraction.

Ports

- Ports serve as message queues for agents.
- A port is a procedure connected to a stream.
- A stream is a variable S that is incrementally constrained to a list by telling constraint for every element of the list:

  $$S = X_1|S_1, S_1 = X_2|S_2, S_2 = X_3|S_3, \ldots$$

  leading to the list $X_1|X_2|X_3| \cdots$
- Ports are implemented in OZ using cells and procedure abstractions.
- A port has a state since it remembers the current tail of its stream.
- Ports are concurrent data structures which allow several concurrent computation to write consistently on a single stream.

```
proc {Newport Stream Port}
   local Cell in
     {NewCell Stream Cell}
     proc {Port Message}
        local Old New in
          {Exchange Cell Old New} Old=Message|New
end end end end
```

- The application {Port X} will tell the constraint $S_i = X|S_{i+1}$ where S_i is the current tail of the stream and S_{i+1} is a new variable for the new tail of the stream.

Agents

An agent is a computational abstraction that:
processes messages received through ports
encapsulates an internal state
may send messages to other agents.
The functionality of an agent is given by the procedure:

$$Serve : State \times Message \rightarrow State$$

describing how the agent serves a message and how it updates its internal state.

```
proc {NewAgent Serve Init Port}
   local Stream Feed in
      {NewPort Stream Port}
      {Feed Stream Init}
      proc {Feed Ms State}
         if Message Mr NewState in Ms=Message|Mr then
           {Serve State Message NewState}
           {Feed Mr NewState}
         else true fi
      end
   end
end
```

Creates a new agent that receives messages through Port and operates as specified by the procedure Serve starting from the initial state Init.

Example: Queue Agent

A queue agent receiving messages through the port Q can be created with:

```
local Xs in {NewAgent QueueServe Xs|Xs Q}
```

where the procedure QueueServe is defined as:

```
{NewName Enqueue}
{NewName Dequeue}
proc {QueueServe State Message NewState}
   if First Last in State=First|Last then
      if X NewLast in Message=Enqueue|X then
         Last=X|NewLast
         NewState=First|NewLast
      else
        if X NewLast in Message=Dequeue|X then
           First=X|NewFirst
           NewState=NewFirst|Last
        else true fi
      fi
    else true fi
end
```

Messages are represented as pairs Enqueue|X and Dequeue|X, where the variables Enqueue and Dequeue are bound to names identifying the corresponding operations.
In the case of an empty queue, a dequeue request will wait in a queue of unserved requests until an item will be entered in the queue.

Objects

- Object are a modular programming abstraction for concurrent data structures with state.
- Object can be modeled in kernel OZ as procedures of type:

  $$Serve : State \times Message \times Self \rightarrow State$$

  defined by:

```
proc {NewObject Serve Init Object}
   local Cell in
      {NewCell Init Cell}
      proc {Object Message}
        local State NewState in
          {Exchange Cell State NewState}
          {Serve State Message Object NewState}
end end end end
```

- The *states of objects* are modeled as finite mappings from attributes (i.e. names) to variables.
- *Methods* are modeled as serve procedures. Methods can construct new states from given state assigning variables to attributes. *Private methods and attributes* by means of lexical scoping and names.
- *Classes* can be modeled defining Serve procedures in a modular fashion by means of named methods. *Inheritance* by means of higher-order functional programming.

Computation Spaces, Actors and Blackboards

- The notion of task is generalized by *actors*.
- A computation space consist of a number of actors connected to a *blackboard* (e.g. the shared store).
- Actors may spawn *local computation spaces* containing other actors and a local blackboard.
- Constraints and abstractions from the parent blackboard are propagated to the local blackboard.
- A (local) computation space disappears when either:
  - all their actors reduce to constraints that are consistent with the blackboard and the resulting constraints are conjoined to the parent blackboard, or
  - one of their actor reduces to a constraint which is inconsistent with the blackboard; in this case the computation space is marked as *failed*.
- Actors can be of the following types:
  - elaborators
  - conditionals
  - disjunctions.

Actor types and reductions

- Actors are defined by expressions given by:

$$\begin{array}{lcll} E & ::= & \varphi \mid E_1\, E_2 \mid \textbf{local}\, x_1, \ldots x_n\, \textbf{in}\, E\, \textbf{end} & \\ & \mid & \textbf{proc}\{x\, y_1 \ldots y_n\}\, E\, \textbf{end} \mid \{x\, y_1 \ldots y_n\} & elaborators \\ & \mid & \textbf{if}\, \varphi\, \textbf{then}\, E_1\, \textbf{else}\, E_2 & \\ & \mid & \textbf{if}\, C_1 \Box \cdots \Box C_n\, \textbf{else}\, E\, \textbf{fi} & conditionals \\ & \mid & \textbf{dis}\, C_1 \Box \cdots \Box C_n\, \textbf{end} & disjunctions \end{array}$$

  where φ denotes a constraint.

- Elaborators reduce as usual.

- C_i are clauses of the form:

$$C_i ::= E_1^i\, \textbf{then}\, E_2^i \mid x_1^i, \ldots, x_n^i\, \textbf{in}\, E_1^i\, \textbf{then}\, E_2^i$$

Conditionals actors and clauses

- $\textbf{if}\, \varphi\, \textbf{then}\, E_1\, \textbf{else}\, E_2$ waits until the blackboard entails either φ or $\neg\varphi$ and then reduces accordingly to E_1 or E_2.

- $\textbf{if}\, C_1 \Box \cdots \Box C_n\, \textbf{else}\, E\, \textbf{fi}$ spawn n local computation spaces each containing the clause C_i.
  - Each clause has the form:

$$C_i ::= E_1^i\, \textbf{then}\, E_2^i \mid x_1^i, \ldots, x_n^i\, \textbf{in}\, E_1^i\, \textbf{then}\, E_2^i$$

    where the local variables $x_1^i, \ldots, x_n^i$ range over the guard E_1^i and the body E_2^i.
  - A guard which is not a constraint is called *deep*.
  - An empty blackboard is created, the parent constraints are propagated and the clause is transformed into the actor: $\textbf{local}\, x_1^i, \ldots x_n^i\, \textbf{in}\, E_1^i\, \textbf{end}$.
  - A clause C_i is *entailed* if
    * its associated computation space S_i is not failed,
    * S_i has no actors left (unreduced) and
    * the global board entails $\exists \overline{y^i} \varphi_i$, where $\overline{y^i}$ are the local variables of S_i and φ_i is the contained constraint.
  - A conditional actor must wait until:
    * either one of its clauses C_i is entailed: the other clauses (and spaces) are discarded, S_i is merged with the global space and the conditional actor reduces to the elaborator E_2^i.
    * or all its clauses are failed: the conditional actor reduces to elaborator E.

Disjunctions (non-determinism)

- A disjunctive actor $\textbf{dis}\, C_1 \Box \cdots \Box C_n\, \textbf{end}$ spawns a local computation space for each clause C_i. Constraints and named abstractions are propagated downward from global to local blackboards.

- A disjunctive actor must wait until:
  - either all but possibly one of its clauses are failed:
    * if all clauses are failed, the space of disjunctive actor is failed.
    * if all but one clause fail:
      · the disjunctive actor reduces to the the body of the unfailed clause (*unit commit*).
      · S_i is merged with the global space (provided that it results into a consistent blackboard, otherwise the global space fails).
  - or a clause whose body is the trivial constraint $true$ is entailed: the actor disappears.

Example (1)

We assume OZ has $\mathcal{RT}$ built-in constraints:

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

Let consider a variant of the `local` expression such that it introduces variable whose scope extends to expressions the programmers enters later:

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

This expression spawn a disjunctive actor that cannot be reduced since there is no enough information on the global blackboard.
Suppose we enter the constraint:

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

Since it is inconsistent with N=0, then the first clause fails and the disjunctive actor is reduced to the second clause:

```
local X Xr M in Xs=X|Xr N=s(M) then {Length Xr M}
```

which in turn assert the constraint S:

```
∃X,Xr,M:(Xs=X|Xr ∧ N=s(M)∧ N=s(s(_)))
```

and elaborate the application of Length whose first clause immediately fails since: N=s(s(_))∧ N=S(M).

Example (2)

This create a new disjunctive actor:

```
dis Xr'=nil M'=0 then true
[] X'' Xr'' M'' in Xr'=X''|Xr'' M'=s(M'') then {Length Xr'' M''}
end
```

asserting

$$\exists X,Xr,M:(Xs=X|Xr \wedge N=s(M) \wedge N=s(s(\_)) \wedge \exists X',Xr',M':(Xr=X'|Xr' \wedge M=s(M')))$$

which this time cannot reduce since no one of the two clauses fails.

Now we enter the constraint:

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

whose elaboration fails the second clause since the current blackboard entails:

$$\exists X,Xr,X',Xr',X'' Xr'': (Xs=X|Xr \wedge Xr=X'|Xr' \wedge Xr'=X''|Xr'')$$

that is Xs=_|_|_|_ (a list of at least 3 elements).

The suspended actor reduces to the first clause asserting:

$$\exists X,Xr,M:(Xs=X|Xr \wedge N=s(M) \wedge N=s(s(\_)) \wedge \exists X',Xr',M':(Xr=X'|Xr' \wedge M=s(M') \wedge M'=0)$$

that finally entails N=s(s(0)).

Encapsulated search

- PROLOG allows to enumerate all the Xs, N such that $length(Xs, N)$ is satisfied. Solutions can be enumerated on-demand by the user interface (or with set-oriented built-in predicates like bagof).

- OZ provides search agents that can be given queries and be prompted for answers.

- A Search agent can be viewed as an object which looks for solution on-demand.

Suppose that Search is a predefined search object (which can be specialized using inheritance). Then a query for the previous example can be activated by executing the method query of the Search object (using functional notation):

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

The first solution is obtained calling the method next:

```
{Search next}
```

returns the pair nil|0. Sending next once more will produce (_|nil)|s(0), and so on.

Sending the message action(P) to the Search object, the unary procedure P can be applied to every solution found.