Development of Concurrent Constraint Languages for multiparadigm programming

Seif Haridi

Swedish Institute of Computer Science
Box 1263, S-164 28 Kista, Sweden
tel +46-08-752 15 30, fax +46-08-751 72 30
e-mail seif@sics.se

FGCS'94 December 1994

This Talk

CCP LANGUAGES

- Basic CCP
- Deep CCP (AKL)
- Higher Order CCP (Oz)
- Integration of Search, Concurrent Objects and Higher Order

Many thanks to fiends from SICS and DFKI, specially Sverker Janson and Johan Montelius SICS and Gert Smolka and Christian Schulte DFKI

Concurrent Constraint Programming

- combines ideas from
  - concurrent logic programming
  - constraint logic programming
  - functional programming
  - concurrency theory

- aims at
  - concurrent symbolic programming
  - problem solving with constraints and search
  - combination of the above
  - parallel implementation
  - transparent distribution

CCP Important developments

Flat CCP

- FGHC ICOT (Ueda, Chikayama) 1985
- Michael J. Maher. Logic semantics for a class of committed-choice programs. 1987.
- Vijay A. Saraswat. Compositional Syntax, the notion of constraint store, ask and tell operations.
- Yang/Warren/Haridi. Introduction of nondeterminism and global search. The notion of determinism driven execution

Deep CCP

- Haridi/Janson. AKL (Agents Kernel Lang). Model for guard execution, integration of search and concurrency, the notion of stability, encapsulated search.
- AKL, ports (named streams), essential for integration of concurrent object oriented programming, and encapsulation of mutable state.

CCP Important developments 2

- Smolka et al, DFKI, Oz. Higher order CCP. Record datatypes, names: subsumes functional programming, modules as special case of usage of records and HO CCP.
- Oz, Cells as simplification of AKL/ports, concurrent object-oriented programming based on prototypes.
- Oz, controlled encapsulated search using the solve combinator. Search is programmable at the user level.

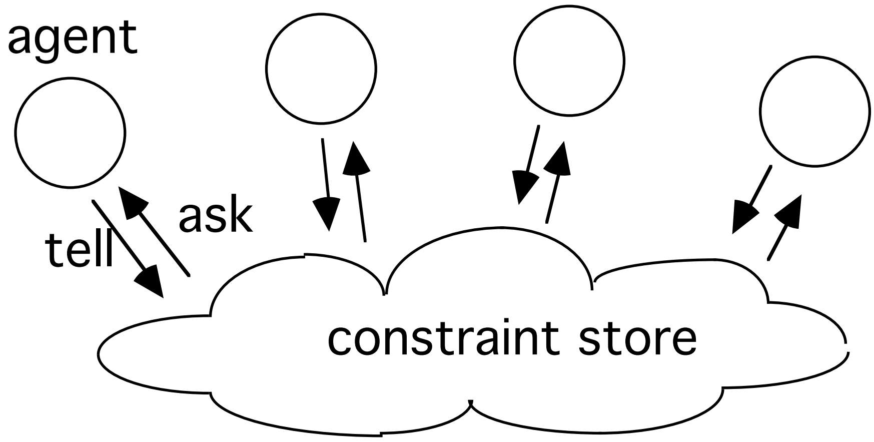
Future

AGENTS II, SICS/DFKI

Adds: Distribution aspects (language for distributed applications)

- persistence, real-time, groups for concurrent exception handling, continuous operation and resource management.
- efficient implementation

CCP



Agents *tell* constraints to a shared constraint store and may *ask* if constraints are entailed by the store.

A flat language (cc flavour)

E	::=	ϕ	*constraint*
	|	$p(\bar{x})$	*call*
	|	$E_1 \quad E_2$	*composition*
	|	$\bar{x}$ **in** E	*hiding*
	|	**or** C_1 | ... | C_n **end**	*disjunction*
	|	**if** C_1 | ... | C_n **else** E **end**	*conditional*

C	::=	$\bar{x}$ **in** $\phi \rightarrow E$	*clause*

D	::=	**proc** $p(\bar{x})$ E **end**	*definition*

C **entailed** by ψ iff ψ implies $\exists \bar{x} \phi$

C **disentailed** by ψ iff ψ implies $\neg \exists \bar{x} \phi$

Reduction Rules Flat CCP

- $\phi \Rightarrow$ tell ϕ and halt if store is unsatisfiable

- $E_1\ E_2 \Rightarrow E_1 \wedge E_2$

- $\bar{x}$ **in** E $\Rightarrow$ E[$\bar{y}/\bar{x}$] if $\bar{y}$ are fresh new variables.

- p($\bar{y}$) $\Rightarrow$ E[$\bar{y}/\bar{x}$] if **proc** p($\bar{x}$) E **end** is in program

proc {p $\overline{\mathbf{X}}$} E end

- **if** C_1 | ... | C_n **else** E **end** $\Rightarrow \langle C_k \rangle$ if C_k entailed

- **if** C_1 | ... | C_n **else** E **end** $\Rightarrow$ E if C_1 **,... ,**C_n disentailed

$\langle\, \bar{x}$ **in** $\phi \rightarrow E \,\rangle \equiv \bar{x}$ **in** $(\phi\ E)$

Examples

This language is basically KL1 if the domain is trees:

```
proc append(xs, ys, zs)
if   xs = [] → zs = ys
|    x,xr,zr in xs = [x | xr] →
     zs = [x | zr]
     append(xr,ys,zr)
end
end
```

or in functional notations

```
fun append(xs, ys)
if   xs = [] → ys
|    x,xr in xs = [x | xr] →
     [x | append(xr,ys)]
end
end
```

Examples 2

```
proc merge(xs, ys, zs)
if   xs = [] → zs = ys
|    ys = [] → zs = xs
|    x,xr,zr in xs = [x | xr] →
     zs = [x | zr]
     merge(xr,ys,zr)
|    y,yr,zr in ys = [y | yr] →
     zs = [y | zr]
     merge(xs,yr,zr)
end
end
```

Reduction of Disjunction

or C_1 | ... | C_n **end** $\Rightarrow \langle C_k \rangle$ if

all C_i different from C_k are disentailed

```
proc length(xs, n)
or   xs=[] n=0 → true
|    x,xr,nr in
     xs=[x|xr], n=S(nr) →
     length(xr, nr)
end
end
```

will be determinacy driven:

xs=[x1,x2,x3] length(xs, n)

produces n=S(S(S(0)))

length(xs, n) n=S(S(S(0)))

produces xs=[x1,x2,x3]

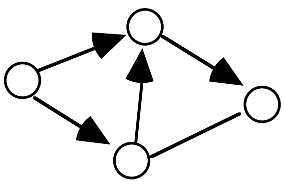
NONDETERMINISM

or C_1 | **...** | C_n **end** , E $\Rightarrow$

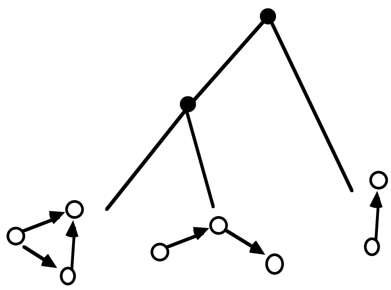
$$(\langle C_1 \rangle \wedge E) \vee \ldots \vee (\langle C_n \rangle \wedge E)$$

if no other reduction is possible

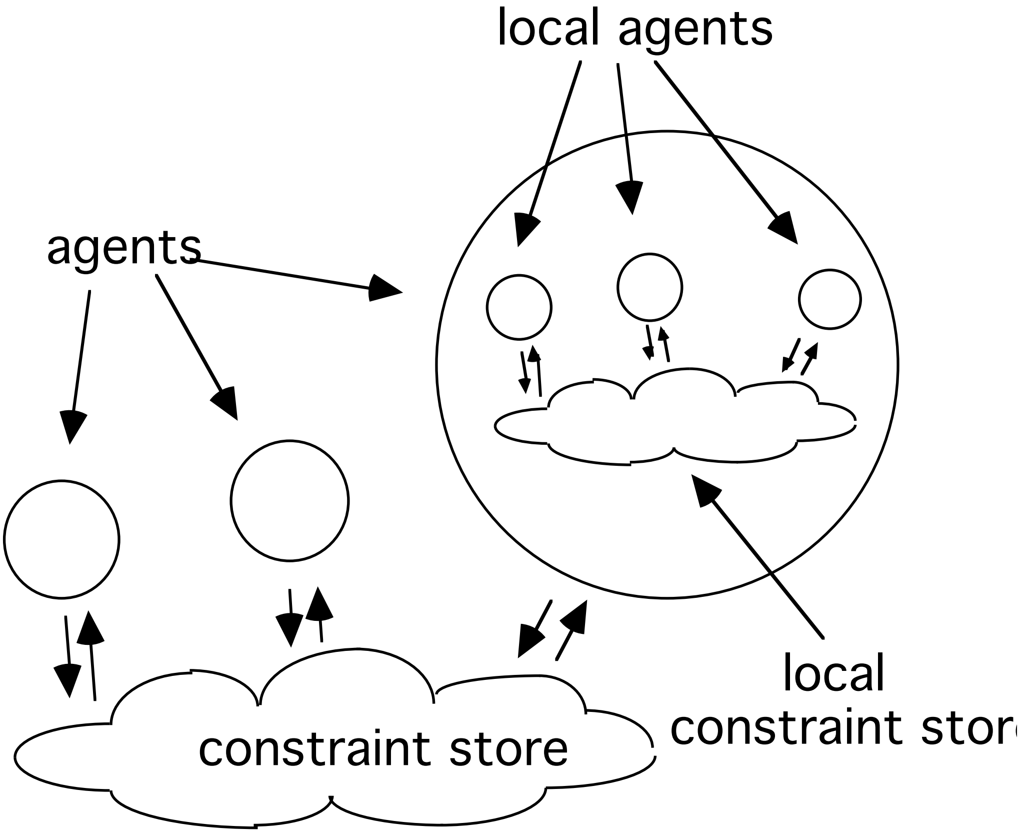
flat+det:



flat+nondet:



Deep CCP

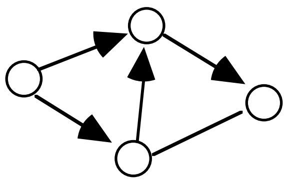


- Local agents see local and external constraints.
- Scope of nondeterminism is local.
- Enables encapsulated search with reactive top-level.
- Supports negation and general conditions.
- The AGENTS implementation of AKL offers record constraints and FD constraints.

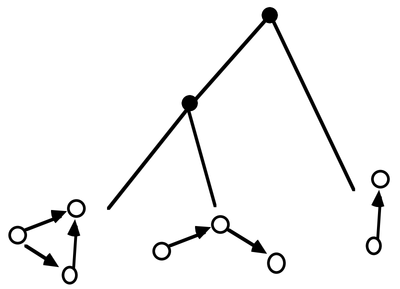
Sverker Janson. *AKL—A Multiparadigm Programming Language*. SICS Dissertation Series 14, Swedish Institute of Computer Science, 1994.

Flat vs deep

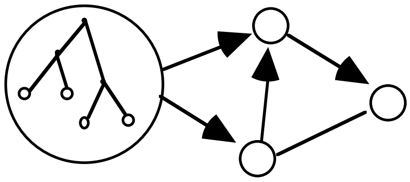
flat+det:



flat+nondet:



deep+nondet:



	"det"	nondet
flat	KL1, Janus, ...	cc, Andorra-I Pandora, ...
deep	CP, GHC, Parlog, ...	AKL, Oz

Deep CCP (AKL)

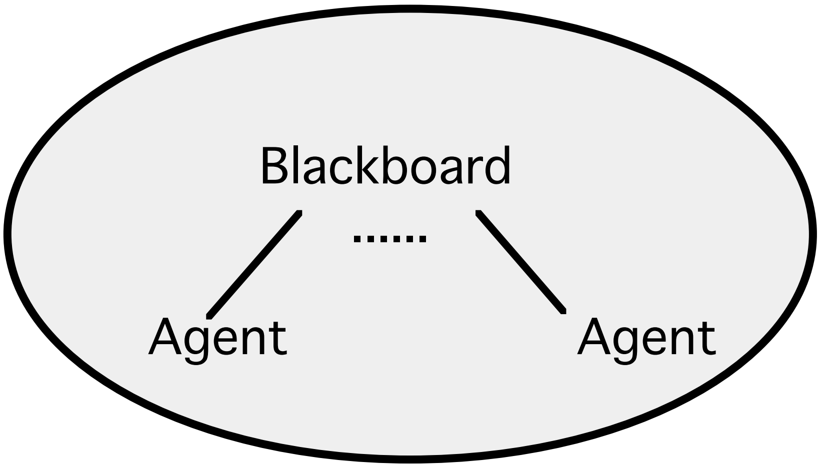
E	::=	ϕ	*constraint*
	|	$p(\bar{x})$	*call*
	|	$E_1 \quad E_2$	*composition*
	|	$\bar{x}$ **in** E	*hiding*
	|	**or** C_1 | ... | C_n **end**	*disjunction*
	|	**if** C_1 | ... | C_n **else** E **end**	*conditional*
	|	**bagof**(x, E_1, y)	*aggregate*

C ::= $\bar{x}$ **in** $E_1 \rightarrow E_2$ *clause*

D ::= **proc** $p(\bar{x})$ E **end** *definition*

not E $=_{def}$ **if** E **then false else true end**

Computation Spaces



- Blackboard

  - Stores expressions to be executed

  - Stores constraints and local variables

  - Expressions are reduced to constraints

  - Information increases monotonically

- Agents associated with expressions

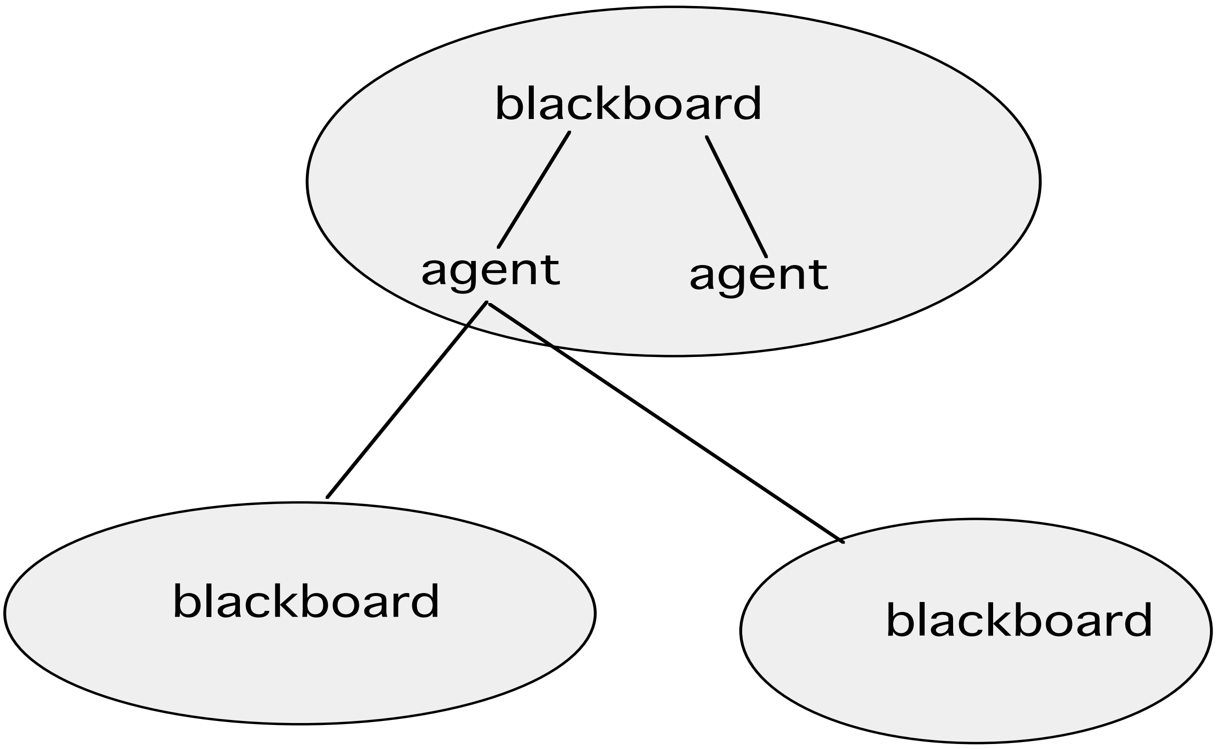
  - read information from blackboard

  - reduce when sufficient information is available

  - upon reduction might add new constraints and spawn new agents

Local Computation Spaces

- Agents may have local computation spaces.
- Information local to a local space is invisible outside.
- Information at higher computation spaces are visible to local computation spaces.



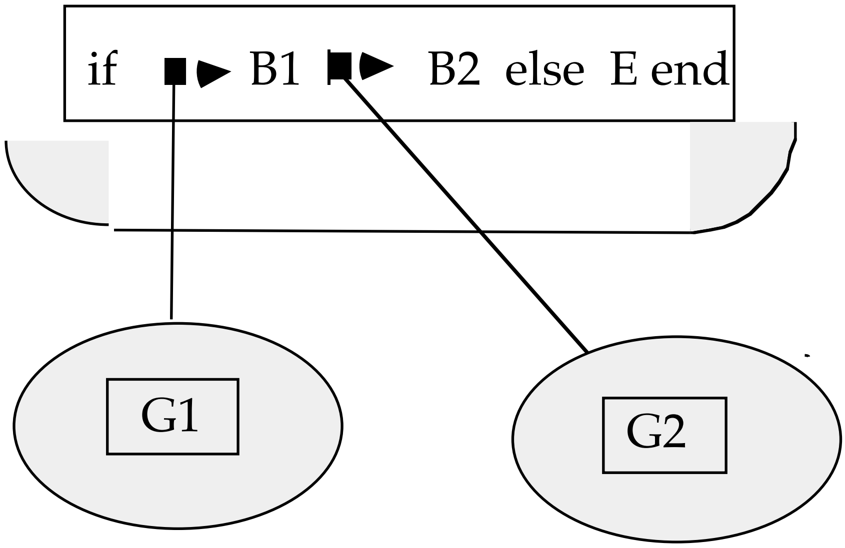
Semantics 3 (Deep)

Conditional expression:

- **if** C_1 | **...** | C_n **else** E **end**

  where

  C_i ::= $\overline{x}_i$ **in** $G_i \rightarrow B_i$



- Spawn a local computation space for each G_i
- Suspend the conditional agent
- Start an agent for each G_i

Stability

An expression E is *stable* if

For all satisfiable constraints ϕ,
$\phi \wedge E$ is not reducible with 'determinate' rules.

A nondeterminate step is allowed in a stable expression.

NONDETERMINISM (choice splitting)

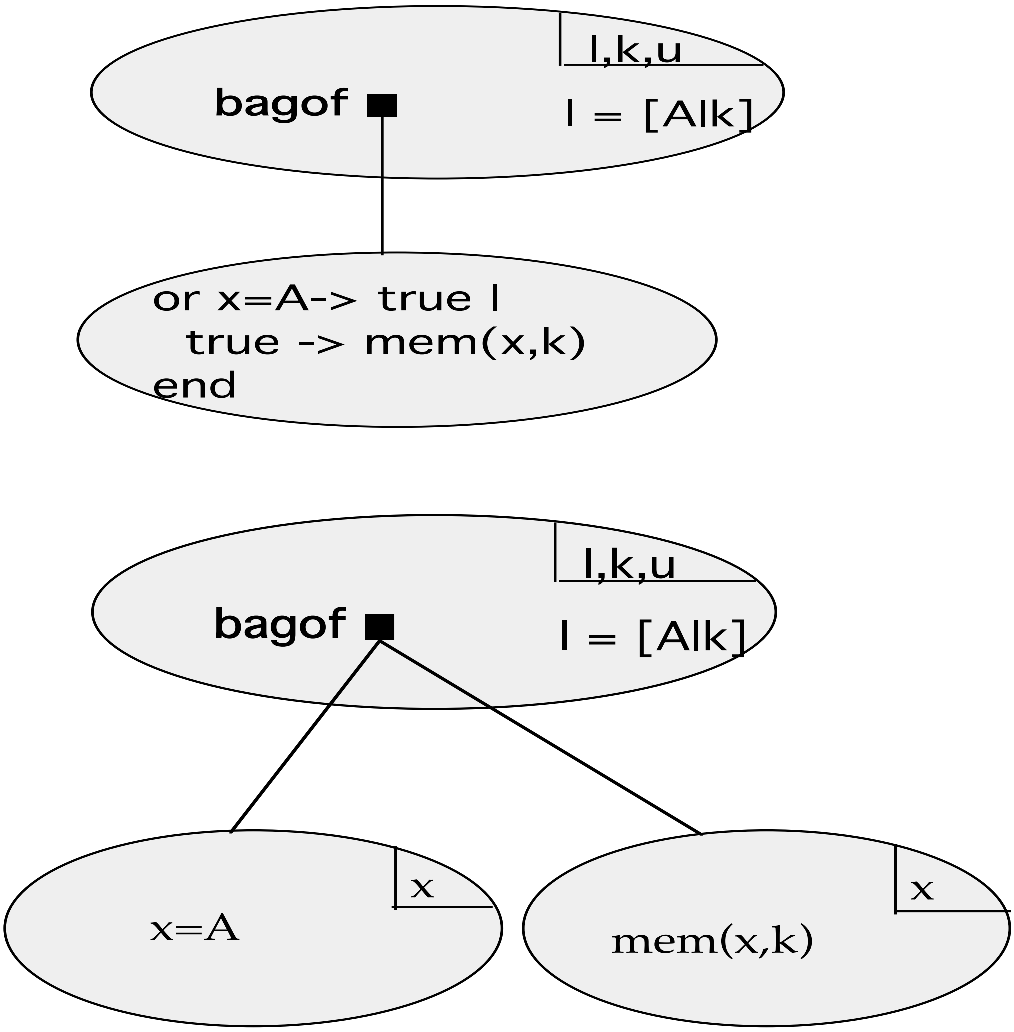
or C_1 | **...** | C_n **end** , E_r $\Rightarrow$

$$(C_1 \wedge E_r) \vee \ldots \vee (C_n \wedge E_r)$$

if the computation space is stable

- The computation space is copied into several copies.
- If the top level computation space is used for external communication then it is always unstable and therefore choice splitting is not allowed.

Example (stability)



l,u,k **in** **bagof**(x,mem(x,l),u)   l=[A | k]

proc mem(x,l)
  or l = [x | _] → true
  | l_1 **in** l = [_ | l_1] → mem(x,l_1)
  end
end

Higher-Order CCP (Oz)

E	::=	ϕ	*constraint*
|		**proc** y($\overline{x}$) E **end**	*definition*
|		y($\overline{x}$)	*call*
|		E_1 E_2	*composition*
|		$\overline{x}$ **in** E	*hiding*
|		**or** C_1 | ... | C_n **end**	*disjunction*
|		**if** C_1 | ... | C_n **else** E **end**	*conditional*
|		**solve**(x:E_1, y)	*solve*

C ::= $\overline{x}$ **in** $E_1 \rightarrow E_2$ *clause*

- Subsumes λ-calculus.
- Elegant support for modules and objects.
- Enables exciting new view of search.
- Introduces a the notion of names in addition to variables (γ calculus)

Higher order (functional) programming

```
fun map(xs, p)
    if   xs = [] → []
    |    x,xr in xs = [x | xr]  →
    [p(x)  | map(xr, p)]
     end
end
```

- Procedures are first class citizens
- Lexical scoping and local definitions
- Work on partial information (incremental)

Objects (Ports)

• AKL introduces the notion of ports:

Persistent references (names) to the end of a stream:

Basic port operations

• new_port(*port*, *stream*)

port is a new name associated with *stream*

• send(*msg*, *port*)

appends *msg* to the stream associated with *port*
reassociate port with the tail of the stream.

Derived port operations

• send(*msg*, *p0,p1*)

p1 is bound to p0 after *msg* is appended to the stream associated with p0

Basic object-oriented style:

```
proc object(port)
local state, stream in
    create_state(state)
    open_port(port, stream)
    obj(stream, state)
end
end

proc obj(ms, state0)
if  msg, mr, s1 in ms = [msg | mr] →
    dispatch(msg,state0, state1)
    obj(mr,state1)
else true
end
```

Properties (Ports)

• Objects have unique identifiers given by their associated port.

• Objects are active concurrent entities, e.g. in the sense of actors.

• The state is serialised and is not available to the next message until the dispatch operation is performed.

• State is encapsulated.

Concurrent Objects (Style)

- class is a record structure its features are the name of messages and its values are the corresponding methods:

  $Class(msg_1:method_1, \dots , msg_n:method_n)$

example

```
   Class(Inc:
            proc (msg, s, ns)
                  x=s.Val, adjoin(Val:x+1,s,ns)
            end)

proc object(o, class)
   local
      proc generic_object(str, state)
         local msg, str1, l in
      if str = [msg | str1] →
                  label(msg,l)
                  (class.l)(msg,state,new_state)
                  generic_object(str1, new_state)
            end
         end
      end
      stream, state in
            new_port(o,stream)
            create_state(state,class,o)
            generic_object(stream, state)
   end
end
```

Object (Sugared Syntax)

```
object counter
    from ur_object
    attrs   val  with Set(0) end
    meth Set(v)  val ← v  end
    meth Inc  val ← @val + 1 end
    meth Get(v) v ← @val end
end

Inc^counter
Inc^counter
```

Properties of Object System

- Encapsulated State
  - implementation optimised due to single reference property
- Multiple inheritance/differential
- Method delegation
- Private methods and attributes (due to local names)
- Meta-object protocol
  - constraints on message acceptence can be inherited and/or refined
- Methods
  - as first class objects
  - state threading
  - late binding (default)

Controlled Encapsulated Search

C. Schulte, G. Smolka, and J. Würtz. Encapsulated Search and Constraint Programming in Oz. In *Principles and Practice of Constraint Programming 1994*.

- Search should not be first principle
- Different strategies should be expressible, including one solution, all solutions, best solution (B&B), and demand driven search
- Problem specification (declarative) and search strategy (operational) should be discribable by different programs

Solve Combinator

solve(x:E, u) ⇒

Failed

solve(x:false, u) ⇒ u=Failed

Solved

solve(x:E, u) ⇒ u = Solved(**proc** (x) E **end**)

Stable

solve(x : (**or** A | B **end** , C), u) ⇒

Distributed(**proc** (x) A C end, **proc**(x) B C **end)**

Depth-First Search

```
proc depth(q, u)
  local v in
     solve(q, v)
     if    v = Failed → u = Failed
     |      s in v = Solved(s)  → u = Solved(s)
     |      l,r,v1 in v = Distributed(l,r) →
               depth(l, v1)
               if v1 = Failed  →
                  depth(r, u)
               else u = v1
               end
     end
  end
end
```

One Solution Search

```
proc one(q, u)
  local v in
     solve(q, v)
     if   v = Failed → u = Failed
     |     s in v = Solved(s)  → u = Solved(s)
     |     l,r,v1 in v = Distributed(l,r) →
           if    s in one(l, Solved(s))  →
                 u = Solved(s)
           |     s in one(r, Solved(s))  →
                 u = Solved(s)
           else
                 u = Failed
           end
     end
  end
end
```

Bagof Aggregate Search (Ordered)

```
proc bagof(q, a)
    local
        proc bagof1(q, u,w)
            local v in
            solve(q, v)
            if   v = Failed → u = w
            |     s in v = Solved(s)  →
                  x in s(x), u = [x | w]
            |     l,r,m in v = Distributed(l,r) →
                  bagof1(l, u, m)
                  bagof1(r, m, w)
            end
        end
    end
    in bagof(q, a, [])
end
```