Computational Logic
Concurrent (Constraint) Logic Programming
Concurrent Logic Programs

- **Predicate:** Set of clauses
- **Clause:** $Head :\neg Guard \mid Body$.
  - $Head$ is an atom
  - $Guard$ and $Body$ are conjunctions of atoms
- **Resolvent:** Set of goals (instances of atoms)
- **Operational semantics:** rewrite a goal in the resolvent with one of the clauses in the matching predicate definition
- **Concurrency:**
  - “No” goal selection rule (i.e., concurrent selection rule)
  - “No” clause search rule (i.e., concurrent search rule)
Synchronization Rules

- Clause matching: $Head + Guard$.
  - $Head$ matches the goal
  - $Guard$ is successful
- A head matches a goal if the goal is an instance of the head
- A guard is executed in one-way unification mode
- Suspension: if a head does not match the goal, but it could do so in the future, then it suspends
An Example

\[ p(X) : \neg X = a \mid r. \]
\[ p(X) : \neg X = b \mid s. \]
\[ q(X) : \neg \text{true} \mid X = b. \]

\[ ?- p(X), q(X). \]

- There is no ordering in the execution of \( \langle p(X), q(X) \rangle \)
- There is no ordering in the execution of clauses of \( p(X) \)
- Clauses of \( p(X) \) suspend
- The clause of \( q(X) \) continues ("commits")
- Then, \( q(X) \) instantiates \( \{X/b\} \) in the body
- The second clause of \( p(X) \) continues ("commits"), while first clause fails.
Logic vs. Concurrent Logic Programming

• The logical variable as a communication channel

<table>
<thead>
<tr>
<th>Logic</th>
<th>Concurrent Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared logical variable</td>
<td>communication channel</td>
</tr>
<tr>
<td>instantiation</td>
<td>communication</td>
</tr>
<tr>
<td>head unification</td>
<td>synchronization</td>
</tr>
</tbody>
</table>

• **Unification Revisited:**
  
  ◇ One-way (Read-only) unification — **Ask**
  
  * in *Head* and in *Guard*

  ◇ Two-way (Output) unification — **Tell**
  
  * only in *Body*

  ◇ Suspension:
  
  * Due to read-only unification in clause selection
Logic vs. Concurrent Logic Programming

- Committed-choice: clause selection is irrevocable
- No backtracking allowed

<table>
<thead>
<tr>
<th>Logic</th>
<th>Concurrent Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>cut</td>
<td>commit</td>
</tr>
<tr>
<td>“don’t know” non-determinism</td>
<td>(“don’t care” non-determinism) indeterminism</td>
</tr>
<tr>
<td>search</td>
<td>selection</td>
</tr>
</tbody>
</table>

- Guards:
  - Flat guards: only selected predicates in guards
    * (Special) builtins
    * Possibly also facts
  - Deep guards: calls to any predicate allowed in guards
    * User-defined predicates, too
Logic vs. Concurrent Logic Programming

- Goals as processes:

<table>
<thead>
<tr>
<th>Logic</th>
<th>Concurrent Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic goal</td>
<td>process</td>
</tr>
<tr>
<td>goal (set of atoms)</td>
<td>process network</td>
</tr>
<tr>
<td>clause</td>
<td>process instruction</td>
</tr>
</tbody>
</table>

- Process Behaviour:
  - Change state of process network:
    - Become a new process:
    - Become $k$ concurrent processes:
  - Halt:
  - Change state of data:

- Some syntactic sugar:
  - $A \vdash G | true. \iff A \vdash G | .$
  - $A \vdash true | G. \iff A \vdash | G. \iff A \vdash G.$
  - $A \vdash true | true. \iff A.$
Process Behaviour Examples

- Become a new process: \( A :\text{-} G | B \).
  
  \[ p(X) :\text{-} X= f(a, Y) | q(Y) \].

- Become \( k \) concurrent processes: \( A :\text{-} G | B_1 \ldots B_k \).
  
  \[ p(X) :\text{-} X= f(A, B, C) | q(A), r(B), s(C) \].

- Halt: \( A :\text{-} G | . \)
  
  \[ p(X) :\text{-} X= f(a) | . \]

- Change state of data: \( A :\text{-} G | \ldots A \).
  
  \[ p(X) :\text{-} X= f(a, Y) | Y= f(b, Z), p(Z) \]
  
  \[ p(I, S) :\text{-} I= [H|NI], \text{int}(H) | NS \text{ is } S+H, p(NI, NS) \].
Incomplete Messages

- Back-communication:

  \[ ?- q(X), p(X). \]

  \[ p(X):- X=f(a,Y), check(Y). \]

  \[ check(ok). \]

  \[ q(f(X,Y)):- X=a | Y=ok. \]
Incomplete Messages (Contd.)

- Dialogue:

  \[\text{- q}(X), \text{p}(\text{more}(X)).\]

  \[\text{p}(\text{more}(X)) :\text{- } X = f(a,Y), \text{p}(Y).\]
  \[\text{p}(\text{more}(X)) :\text{- } X = f(b,Y), \text{p}(Y).\]
  \[\text{p}(\text{ok}).\]

  \[\text{q}(f(X,Y)) :\text{- } X = b \mid Y = \text{more}(Z), \text{q}(Z).\]
  \[\text{q}(f(X,Y)) :\text{- } X = a \mid Y = \text{ok}.\]

- Network formation and reconfiguration:

  \[\text{- q}(A), \text{p}(A).\]

  \[\text{p}(A) :\text{- } A = \text{channels}(X,Y,Z), \text{p1}(X), \text{p2}(Y), \text{p3}(Z).\]

  \[\text{q}(\text{channels}(X,Y,Z)) :\text{- } \text{q1}(X), \text{q2}(Y), \text{q3}(Z).\]
The Logical Variable

- A shared variable acts like:
  - A communication channel to send a message
  - A shared location being accessed concurrently

- Equivalences/conceptual view:
  - One shared variable = One message
  - Instantiation = Sending a message
  - Partially instantiated term = incomplete message = open channel
  - Ground term = complete message = closed channel
  - Recursive term = stream of messages

- Incomplete structures: an incomplete message can be thought of as:
  - A message being incrementally sent
  - An open communication channel
  - A message with sender’s identity
  - A structure being co-operatively constructed
Streams of Messages

• A stream producer
  
  \[
  \text{naturals}(N, \text{Is}) :- \text{Is}=[N|\text{Is}1], N1 \text{ is } N+1, \text{naturals}(N1, \text{Is}1).
  \]

• A stream consumer
  
  \[
  \text{sum}([N|\text{Is}], \text{Tmp}, \text{Sum}) :- N \geq 0 \mid \text{TN is } \text{Tmp}+N, \text{sum}(\text{Is}, \text{TN}, \text{Sum}).
  \]

• Producer/Consumer (asynchronous)
  
  ?- \text{naturals}(0, \text{I}), \text{sum}(<\text{I}, 0, \text{Total}).

• Producer/Consumer on demand (synchronous)
  
  ?- \text{naturals}(0, \text{I}), \text{sum}(<\text{I}, 0, \text{Total}), \text{I}=[_|[_].

  \[
  \text{naturals}(N, [\text{I}|\text{Is}]) :- \text{I}=N, N1 \text{ is } N+1, \text{naturals}(N1, \text{Is}).
  \]

  \[
  \text{sum}([\text{N}|\text{Is}], \text{Tmp}, \text{Sum}) :- N \geq 0 \mid \text{Is}=[_|[_], \text{TN is } \text{Tmp}+N, \text{sum}(\text{Is}, \text{TN}, \text{Sum}).
  \]

• Key issue: who produces the buffer?
Merging and Dispatching Streams

- A stream merger:
  
  
  ```
  merge([X|Xs],Ys,Out):- Out=[X|Zs], merge(Xs,Ys,Zs).
  merge(Xs,[Y|Ys],Out):- Out=[Y|Zs], merge(Xs,Ys,Zs).
  merge([],Ys,Out):- Out=Ys.
  merge(Xs,[],Out):- Out=Xs.
  ```

- A (copying) stream dispatcher?
  
  ```
  dispatch([X|Xs],Out1,Out2):- Out1=[X|Ys], Out2=[X|Zs], dispatch(Xs,Ys,Zs).
  dispatch([],Out1,Out2):- Out1=[], Out2=[].
  ```

- A (caotic) stream dispatcher:
  
  ```
  dispatch([X|Xs],Out1,Out2):- Out1=[X|Ys], dispatch(Xs,Ys,Out2).
  dispatch([X|Xs],Out1,Out2):- Out2=[X|Ys], dispatch(Xs,Out1,Ys).
  dispatch([],Out1,Out2):- Out1=[], Out2=[].
  ```

- A stream dispatcher with senders’ identities
  
  ```
  dispatch([mess(1,X)|Xs],Out1,Out2):- Out1=[X|Ys], dispatch(Xs,Ys,Out2).
  dispatch([mess(2,X)|Xs],Out1,Out2):- Out2=[X|Ys], dispatch(Xs,Out1,Ys).
  dispatch([],Out1,Out2):- Out1=[], Out2=[].
  ```
Fairness

“An event that may occur will eventually occur”

- Or-Indeterminism: clause selection $\Rightarrow$ Or-Fairness (clauses eventually selected)
- And-Indeterm.: goal reduction $\Rightarrow$ And-Fairness (allows non-terminating procs.)
- A stream merger:

  ```prolog
  merge([X|Xs],Ys,Out):- Out=[X|Zs], merge(Xs,Ys,Zs).
  merge(Xs,[Y|Ys],Out):- Out=[Y|Zs], merge(Xs,Ys,Zs).
  merge([],Ys,Out):- Out=Ys.
  merge(Xs,[],Out):- Out=Xs.
  ```

  Key: or-fairness required, otherwise it is just append!

- An eager producer:

  ```prolog
  naturals(N,Is):- | Is=[N|Is1], N1 is N+1, naturals(N1,Is1).
  ```

  ?- naturals(0,I), sum(I,0,Total).

  Key: and-fairness required, otherwise nothing is ever consumed!
Termination Issues

- Non–terminating (but running) processes:

  $?- \text{naturals}(I), \text{sum}(I,\text{Total}), I=[\_|\_].$

  \text{naturals}(I):- \text{naturals}(0,I).

  \text{naturals}(N,[I|Is]):- I=N, N1 is N+1, \text{naturals}(N1,Is).

  \text{sum}(I,\text{Total}):- \text{sum}(I,0,\text{Total}).

  \text{sum}([N|Is], \text{Tmp}, \text{Sum}):- N>=0 \mid Is=[\_|\_], TN is Tmp+N, \text{sum}(Is, TN, Sum).
Termination Issues (Contd.)

- Deadlock:

```prolog
?- q(X), p(X).

p(more(X)) :- X=f(a, Y), p(Y).
p(more(X)) :- X=f(b, Y), p(Y).
p(ok).

q(f(X, Y)) :- X=b | Y=more(Z), q(Z).
q(f(X, Y)) :- X=a | Y=ok.
```
Bounded-Size Communication Media

- Producer/Consumer with fixed sized communication (e.g., size=4) and termination:

  ```prolog
  ?- naturals(0,I), sum(I,0,Total), I=[_1,_2,_3,_4].

  naturals(N,[I|Is]):- I=N, N1 is N+1, naturals(N1,Is).
  naturals(N,[]).

  sum([N|Is],Tmp,Sum):- N>=0 | TN is Tmp+N, sum(Is,TN,Sum).
  sum([],Tmp,Sum):- Sum=Tmp.
  ```

  Key: the communication media is produced from outside and fixed size!

- Dynamically-sized media:

  ```prolog
  ?- naturals(0,I), sum(I,0,Total), medium(4,I).

  medium(0,Stream) :- Stream = [].
  medium(N,Stream):- N>0 | Stream=[_|Stream1], medium(N-1,Stream1).
  ```
Bounded-Buffer Communication

- Bounded buffer:
  
  ```prolog
  buffer(0,Stream,Tail):- Stream=Tail.
  buffer(N,Stream,Tail):- N>0 | Stream=[_|Stream1], buffer(N-1,Stream1,Tail).
  ```
  
  Creates buffer as open list of N elements, passes handle to list end

- Simple producer with termination at Max elements:
  
  ```prolog
  naturals(N,[I|Is],Max):- N<=Max | I=N, N1 is N+1, naturals(N1,Is,Max).
  naturals(N,I,Max):- N>Max | I=[].
  ```
  
  Suspended until buffer available. Closes buffer at Max elements

- Consumer:
  
  ```prolog
  sum([N|Is],Tail,Acc,Sum):- N>=0 | 
  Tail=[_|Tail1], NAcc is Acc+N, sum(Is,Tail1,NAcc,Sum).
  sum([],Tail,Acc,Sum) :- Acc = Sum.
  ```
  
  Suspended until buffer and element available. Adds one more element to the buffer for each element consumed.

- Usage (e.g., for buffer length = 18, termination at 1000 elements):
  
  ```prolog
  -: naturals(0,Buffer,1000), sum(Buffer,Tail,0,Total), buffer(18,Buffer,Tail).
  ```
Bounded-Buffer Communication (Contd.)

- Overall effect is still asynchronous!
- Producer can get ahead of consumer by a fixed number of elements. After that, suspended on stream until Consumer requests more.
Streams of Messages: Protocols

- One-to-one communication:
  One producer + One consumer

- Duplex communication:
  Two producer/consumers

- Broadcast communication:
  One producer + Many consumers

- Many-to-one communication:
  Many producers + One consumer

- Blackboard communication:
  Many producers + Many consumers:
  Many producers/consumers
Broadcast Communication

- Matrix multiplication:

\[\text{vm}(_{-}, [], Zv) :- Zv = [].\]
\[\text{vm}(Xv, [Yv|Ym], Zv) :- Zv = [Z|Zv1], \quad \text{vv}(Xv, Yv, Z), \quad \text{vm}(Xv, Ym, Zv1).\]

\[\text{vv}(Xv, Yv, P) :- \text{vv1}(Xv, Yv, 0, P).\]
\[\text{vv1}([], [], S, P) :- P = S.\]
\[\text{vv1}([X|Xv], [Y|Yv], S, P) :- S1 \text{ is } S + X*Y \mid \text{vv1}(Xv, Yv, S1, P).\]

- Broadcasting of \( V \) to all \( \text{vv}/3 \) processes

- Dynamically configured network of \( \text{vv}/3 \) processes
Many-to-one Communication

• A data abstraction: queues

queue([dequeue(X)|S],Head,Tail):-
    Head=[X|NewHead],
    queue(S,NewHead,Tail).
queue([enqueue(X)|S],Head,Tail):-
    Tail=[X|NewTail],
    queue(S,Head,NewTail).
queue([],_,_).
A simulator of a multiprocessor machine

?- processors(10,Job), Job=...

processors(N,X):-
    queue(S,[X|Xs],Xs),
    processors(1,N,S).

processors(N,N,S):-
    processor(N,idle,S).
processors(N1,N4,S):-
    N2 is (N1+N4)/2 | N3 is N2+1,
    processors(N1,N2,S1),
    processors(N3,N4,S2),
    merge(S1,S2,S).

N processor/3 proc. communicating with one queue/3 proc.

Statically configured network of proc.: spawning / computing phases (“systolic”)
Many-to-many Communication

- A network of producers and consumers

```prolog
?- producers(Buffer), consumers(Buffer).

producers(Stream):- p1(X), p2(Y), p3(Z),
  merge(X,Y,Stream1), merge(Z,Stream1,Stream).

consumers(Stream):- c1(Stream), c2(Stream), c3(Stream).

p1(S):- S=[message(1,Mess)|Xs], produce(Mess), p1(Xs).
  p1(S):- S=[].

c1([X|Xs]):- X=message(1,Mess) | consume(Mess), c1(Xs).
  c1([X|Xs]):- X=message(Id,Mess), Id=\=1 | c1(Xs).
  c1([]).
```

- Blackboard Communication:
  - Needed driver for the blackboard
Operational Semantics

- Rewriting system

\[
\text{match}(A, A') = \begin{cases} 
\theta & \text{if } A = A'\theta \text{ and } \text{mgu}(A, A') = \theta \\
\text{fail} & \text{if } \text{mgu}(A, A') = \text{fail} \\
\text{suspend} & \text{otherwise}
\end{cases}
\]

\[
\text{try}(A, (A' \leftarrow G \mid B)) = \begin{cases} 
\theta & \text{if } \text{match}(A, A') = \theta \land \\
& \text{check}(G\theta) = \text{true} \\
\text{fail} & \text{if } \text{match}(A, A') = \theta \land \\
& \text{check}(G\theta) = \text{fail} \lor \\
& \text{match}(A, A') = \text{fail} \\
\text{suspend} & \text{otherwise}
\end{cases}
\]

Operational Semantics (Contd.)

- **Reduction:** \( A_1 \ldots A_i \ldots A_n; \vartheta \rightarrow (A_1 \ldots B_1 \ldots B_k \ldots A_n)\vartheta'; \vartheta \circ \vartheta' \)
  
  if \( \exists C = A \leftarrow G \mid B_1 \ldots B_n \) s.t. \( try(A_i, C) = \vartheta' \)

- **Failure:** \( A_1 \ldots A_i \ldots A_n; \vartheta \rightarrow \text{fail}; \vartheta \)
  
  if \( \forall C \ try(A_i, C) = \text{fail} \)

- **Guard checking:**
  - Flat guards: use \textit{match} in all unifications
  - Deep guards: copy environment
(Some) Concurrent Logic Languages

- Parlog [Clark, Gregory 83]
  - mode declarations for input/output arguments
  - safe clauses: output instantiation in guards is an error
  - one-way unification in guards

- Concurrent Prolog [Shapiro 84]
  - read-only annotation of variables in calls
  - local environments for guards
  - atomic extended head unification

- GHC (Guarded Horn Clauses) [Ueda 85]
  - different interpretation of unification in guard and body
  - suspension on output instantiation in guards
  - general unification with guard restriction
(Some) Concurrent Logic Languages (Contd.)

- Implementation Issues:
  - Parlog
    * compile-time safety check
  - Concurrent Prolog
    * support for local environments
    * detection of inconsistency with global environment
  - GHC
    * identification of variables on which to suspend

- Problems: no backtracking.

- More Recent Systems:
  - Andorra-I: only deterministic computations proceed.
  - AKL: goals execute in a local environment.
  - BinProlog: communication through blackboard.
  - CIAO: communication through shared database.