Computational Logic

Concurrent (Constraint) Logic Programming
Concurrent Logic Programs

- **Predicate**: Set of clauses
- **Clause**: $Head :- Guard | Body$.
  - $Head$ 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: \( \text{Head} + \text{Guard} \).
  - \( \text{Head} \) matches the goal
  - \( \text{Guard} \) 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

\begin{verbatim}
p(X):- X = a | r.
p(X):- X = b | s.
q(X):- true | X = b.

?- p(X), q(X).
\end{verbatim}

- 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”</td>
<td>(“don’t care” non-determinism)</td>
</tr>
<tr>
<td>non-determinism</td>
<td>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

<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>

- **Goals as processes:**

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

- **Some syntactic sugar:**
  - $A \leftarrow G \mid true. \Leftrightarrow A \leftarrow G \mid .$
  - $A \leftarrow true \mid G. \Leftrightarrow A \leftarrow G. \Leftrightarrow A \leftarrow G.$
  - $A \leftarrow true \mid true. \Leftrightarrow A.$

\[ A \leftarrow G \mid B. \]
\[ A \leftarrow G \mid B_1 \ldots B_k. \]
\[ A \leftarrow G \mid true. \]
\[ A \leftarrow G \mid ...A. \]
Process Behaviour Examples

- Become a new process: \( A :\!-\! G \mid B. \)

\[
p(X):= X=f(a,Y) \mid q(Y).
\]

- Become \( k \) concurrent processes: \( A :\!-\! G \mid B_1...B_k. \)

\[
p(X):= X=f(A,B,C) \mid q(A), r(B), s(C).
\]

- Halt: \( A :\!-\! G \mid . \)

\[
p(X):= X=f(a) \mid .
\]

- Change state of data: \( A :\!-\! G \mid \ldots A. \)

\[
p(X):= X=f(a,Y) \mid Y=f(b,Z), p(Z).
p(I,S):= I=[H\mid NI], \text{int}(H) \mid NS \text{ is } S+H, p(NI,NS).
\]
Incomplete Messages

- Back-communication:

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

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

  \[check(ok).\]

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

- Dialogue:

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

- Network formation and reconfiguration:

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

p(A) :- A = channels(X, Y, Z), p1(X), p2(Y), p3(Z).

q(channels(X, Y, Z)) :- q1(X), q2(Y), 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
  \[
  \text{naturals}(N,Is):= Is=[N|Is1], \text{N1 is N+1, naturals}(N1,Is1).
  \]

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

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

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

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

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

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

- A stream merger:
  
  \[
  \text{merge}([X|Xs],Ys,Out):- \quad \text{Out}=[X|Zs], \quad \text{merge}(Xs,Ys,Zs). \\
  \text{merge}(Xs,[Y|Ys],Out):- \quad \text{Out}=[Y|Zs], \quad \text{merge}(Xs,Ys,Zs). \\
  \text{merge}([],Ys,Out):- \quad \text{Out}=Ys. \\
  \text{merge}(Xs,[]Out):- \quad \text{Out}=Xs.
  \]

- A (copying) stream dispatcher?
  
  \[
  \text{dispatch}([X|Xs],Out1,Out2):- \quad \text{Out1}=[X|Ys], \quad \text{Out2}=[X|Zs], \quad \text{dispatch}(Xs,Ys,Zs). \\
  \text{dispatch}([],Out1,Out2):- \quad \text{Out1}=[], \quad \text{Out2}=[].
  \]

- A (caotic) stream dispatcher:
  
  \[
  \text{dispatch}([X|Xs],Out1,Out2):- \quad \text{Out1}=[X|Ys], \quad \text{dispatch}(Xs,Ys,Out2). \\
  \text{dispatch}([X|Xs],Out1,Out2):- \quad \text{Out2}=[X|Ys], \quad \text{dispatch}(Xs,Out1,Ys). \\
  \text{dispatch}([],Out1,Out2):- \quad \text{Out1}=[], \quad \text{Out2}=[].
  \]

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

“An event that may occur will eventually occur”

- Or-Indeterminism: clause selection ⇒ Or-Fairness (clauses eventually selected)
- And-Indetermin.: goal reduction ⇒ And-Fairness (allows non-terminating procs.)
- A stream merger:
  
  \[
  \text{merge}([X|Xs], Ys, Out):- \ Out=[X|Zs], \ \text{merge}(Xs, Ys, Zs).
  \]
  
  \[
  \text{merge}(Xs, [Y|Ys], Out):- \ Out=[Y|Zs], \ \text{merge}(Xs, Ys, Zs).
  \]
  
  \[
  \text{merge}([], Ys, Out):- \ Out=Ys.
  \]
  
  \[
  \text{merge}(Xs, [], Out):- \ Out=Xs.
  \]

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

- An eager producer:
  
  \[
  \text{naturals}(N, Is):- \ | \ Is=[N|Is1], \ N1 \ \text{is} \ N+1, \ \text{naturals}(N1, Is1).
  \]
  
  ?- \ \text{naturals}(0, Is), \ \text{sum}(I, 0, \text{Total}).

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

- Non–terminating (but running) processes:

```prolog
?- naturals(I), sum(I,Total), I=[|\_].

naturals(I):- naturals(0,I).

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

sum(I,Total):- sum(I,0,Total).

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

• Deadlock:

?- 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:
  
  \[
  \begin{align*}
  &\text{buffer}(0, \text{Stream}, \text{Tail}) :- \text{Stream}=\text{Tail}. \\
  &\text{buffer}(N, \text{Stream}, \text{Tail}) :- N>0 \mid \text{Stream}=[_|\text{Stream1}], \text{buffer}(N-1, \text{Stream1}, \text{Tail}).
  \end{align*}
  \]

  Creates buffer as open list of \(N\) elements, passes handle to list end

- Simple producer with termination at Max elements:
  
  \[
  \begin{align*}
  &\text{naturals}(N, [I|\text{Is}], \text{Max}) :- N\leq\text{Max} \mid I=N, N1 \text{ is } N+1, \text{naturals}(N1, \text{Is}, \text{Max}). \\
  &\text{naturals}(N, I, \text{Max}) :- N>\text{Max} \mid I=[].
  \end{align*}
  \]

  Suspended until buffer available. Closes buffer at \(\text{Max}\) elements

- Consumer:
  
  \[
  \begin{align*}
  &\text{sum}([N|\text{Is}], \text{Tail}, \text{Acc}, \text{Sum}) :- N\geq0 \mid \\
  &\quad \text{Tail}=[_|\text{Tail1}], \text{NAcc} \text{ is } \text{Acc}+N, \text{sum}(\text{Is}, \text{Tail1}, \text{NAcc}, \text{Sum}). \\
  &\text{sum}([], \text{Tail}, \text{Acc}, \text{Sum}) :- \text{Acc} = \text{Sum}.
  \end{align*}
  \]

  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):
  
  \[
  \begin{align*}
  &?- \text{naturals}(0, \text{Buffer}, 1000), \text{sum}(\text{Buffer}, \text{Tail}, 0, \text{Total}), \text{buffer}(18, \text{Buffer}, \text{Tail}).
  \end{align*}
  \]
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:

\[ \text{vm}(\text{V}, \text{M}, \text{Result}). \]

\[ \text{vm}(\text{V}, \text{M}, \text{Result}):= \text{vm}(\text{V}, \text{M}, \text{Result}) \]

\[ \text{vm}(_, [], \text{Zv}):= \text{Zv}=[]. \]

\[ \text{vm}(\text{Xv}, [\text{Yv} \text{|} \text{Ym}], \text{Zv}):= \text{Zv}=[\text{Z} \text{|} \text{Zv1}], \]

\[ \text{vv}(\text{Xv}, \text{Yv}, \text{Z}), \]

\[ \text{vm}(\text{Xv}, \text{Ym}, \text{Zv1}). \]

\[ \text{vv}(\text{Xv}, \text{Yv}, \text{P}):= \text{vv1}(\text{Xv}, \text{Yv}, 0, \text{P}). \]

\[ \text{vv1}([], [], \text{S}, \text{P}):= \text{P}=$$ \]

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

\[ \text{vv1}(\text{Xv}, \text{Yv}, \text{S1}, \text{P}). \]

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

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

- A data abstraction: queues

  queue([enqueue(X)|S],Head,Tail):-
      Head=[X|NewHead],
      queue(S,NewHead,Tail).

  queue([dequeue(X)|S],Head,Tail):-
      Head=[X|NewHead],
      queue(S,Head,NewTail).

  queue([],_,_).

  queue(S,Head,NewTail).

  queue(S,Head,NewTail).

  queue(S,Head,NewTail).

  queue(S,Head,NewTail).
Many-to-one Communication (Contd.)

• 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”)

23
Many-to-many Communication

- A network of producers and consumers

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

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

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

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

- **Failure**: $A_1...A_i...A_n; \theta \rightarrow fail; \theta$
  
  if $\forall C \ try(A_i, C) = fail$

- **Guard checking**:
  - Flat guards: use $match$ 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.