Monadic Concurrent Linear Logic Programming

Pablo Lopez	Frank Pfenning	Jeff Polakow	Kevin Watkins
UMA	CMU	AIST	CMU

July 11, 2005

□ LolliMon

Our paper introduces a new logic-programming language, LolliMon, which features:

- Goal-directed, backward-chaining proof search corresponding to serial computation
- Saturation-based, forward-chaining proof search corresponding to concurrent computation
- Linear logic allowing stateful computation

A monad is used to smoothly integrate forward and backward proof search.

□ Outline of Talk

1. Backwards-chaining and Forwards-chaining
2. LolliMon Executions
3. Pi-calculus Example
4. Graph Bipartite Checking Example
5. Conclusion

□ Backward Chaining

- Prolog-style logic programming
  (e.g. Prolog, λProlog, Lolli)

- Goal-directed and Focussed

- Based on asynchronous formulas—
  Right rules can always be safely applied

- Serial computation—
  Atomic goals are function calls

□ Forward Chaining

- Bottom-up logic programming
  (e.g. Datalog, Concurrent Constraints, Logical Algorithms)

- Context-driven

- Based on saturation—
  System computes until it gets stuck

- Concurrent computation—
  Context formulas are processes

□ Combining Paradigms

It is useful to have a system which can use both forward and backward reasoning.

- larger, more expressive formula language
- representation of both concurrent and serial computation

We want a principled way to mix search strategies.

- clean and predictable operational semantics
- computationally useful proof structures

□ Monadic Uniform Proofs

- Uniform proofs (goal-directed, focussed) underlie Prolog-style languages.

- Synchronous formulas destroy uniformity since they are not goal-directed.

- We can combine synchronous formulas and uniform proof structure with a monad, $\{\cdot\}$.
  - Encapsulate treatment of synchronous formulas.
  - Non goal-directed behavior is similar to effects in a functional language.

□ LolliMon Execution

The main branch of a LolliMon execution has the following form:

$$
\begin{array}{lll}
\text{backward-chaining} & \left\{ \begin{array}{l} \Gamma_0 \vdash G_0 \\ \Gamma_0 \vdash G_1 \\ \quad \vdots \end{array} \right. & \text{serial computation} \\
\text{forward-chaining} & \left\{ \begin{array}{l} \Gamma_0 \vdash \{G'\} \\ \Gamma_1 \vdash \{G'\} \\ \quad \vdots \\ \Gamma_n \vdash \{G'\} \end{array} \right. & \begin{array}{c} \text{concurrent computation} \\ \text{(in monad)} \end{array} \\
\text{backward-chaining} & \left\{ \begin{array}{l} \Gamma_n \vdash G' \\ \Gamma_n \vdash G'_0 \\ \quad \vdots \end{array} \right. & \text{serial computation}
\end{array}
$$

Monadic goal signals switch to forward-chaining.
Saturation signals return to backward-chaining.

□ LolliMon Execution– Details

The forward-chaining section of a LolliMon execution has following (simplified) form:

$$
\dfrac{\Gamma' \vdash A_0 \qquad \dfrac{\Gamma' \vdash A_1 \qquad \dfrac{B_0 \vdash B_0 \qquad \dfrac{\dfrac{\Gamma_n \vdash G'}{\Gamma_n \vdash \{G'\}}}{\begin{array}{c}\vdots\\ C_0,\ B_1,\ \Gamma' \vdash \{G'\}\end{array}}}{B_0,\ B_1,\ B_0 \supset \{C_0\},\ \Gamma' \vdash \{G'\}} \supset_L}{B_0,\ A_1 \supset \{B_1\},\ B_0 \supset \{C_0\},\ \Gamma' \vdash \{G'\}} \supset_L}{A_0 \supset \{B_0\},\ A_1 \supset \{B_1\},\ B_0 \supset \{C_0\},\ \Gamma' \vdash \{G'\}} \supset_L
$$

Only monadic-headed clauses used during forward chaining steps.

□ Concurrent Interpretation

New independent subgoals can be executed in parallel:

$$\dfrac{\Gamma' \vdash A_0 \qquad \Gamma' \vdash A_1 \qquad \dfrac{B_0 \vdash B_0 \qquad \begin{array}{c} \dfrac{\Gamma_n \vdash G'}{\Gamma_n \vdash \{G'\}} \\ \vdots \\ C_0,\ B_1,\ \Gamma' \vdash \{G'\} \end{array}}{B_0,\ B_1,\ B_0 \supset \{C_0\},\ \Gamma' \vdash \{G'\}}}{A_0 \supset \{B_0\},\ A_1 \supset \{B_1\},\ B_0 \supset \{C_0\},\ \Gamma' \vdash \{G'\}}$$

Each new subgoal-derivation is an atomic step—
A_0, A_1, B_0 will each only be derived once.

□ LolliMon Computational Interpretation

- Backward-chaining— (Serial Programming)
  - Goal formula is currently executing function.
  - Prolog-style backtracking behavior.
- Forward-chaining— (Concurrent Programming)
  - Process execution denotes an atomic step.
    Incomplete proof search strategy.
  - Forward chaining stops upon saturation.
- Program clauses are unrestricted (intuitionistic) hypotheses.
- Data are (usually) linear hypotheses.

□ Pi-calculus in LolliMon

- Directly interpret Pi-calculus connectives with LolliMon formulas.

- Processes are linear hypotheses.

- Pi-calculus operational semantics are monadic-headed program clauses (i.e. rewrite rules).

- Entirely forward-chaining, execute with $\{\top\}$ as goal:

$$\Delta_I \vdash \{\top\} \quad \leadsto \quad \Delta_O \vdash \{\top\}$$

where Δ_I and Δ_O are the start and stop process states.

□ Pi-calculus signature

```
expr : type.
chan : type.

par  : expr -> expr -> expr.
zero : expr.
new  : (chan -> expr) -> expr.

in   : chan -> (chan -> expr) -> expr.
rin  : chan -> (chan -> expr) -> expr.
out  : chan -> chan -> expr.
```

Interpretation

$$
\begin{array}{lcl}
\ulcorner P \mid Q \urcorner & = & \mathtt{par}\,\ulcorner P \urcorner\,\ulcorner Q \urcorner \\
\ulcorner \bar{c}\langle v \rangle \urcorner & = & \mathtt{out}\ c\ v \\
\ulcorner c(x).P(x) \urcorner & = & \mathtt{in}\ c\ (\ulcorner \lambda x{:}\mathtt{chan}.P(x) \urcorner) \\
\ulcorner !c(x).P(x) \urcorner & = & \mathtt{rin}\ c\ (\ulcorner \lambda x{:}\mathtt{chan}.P(x) \urcorner) \\
\ulcorner \nu c.P(c) \urcorner & = & \mathtt{new}\ (\ulcorner \lambda c{:}\mathtt{chan}.P(c) \urcorner) \\
\ulcorner 0 \urcorner & = & \mathtt{zero}
\end{array}
$$

□ Pi-calculus operational semantics

```
proc : expr -> o.
msg  : chan -> chan -> o.

proc (par P Q) -o {proc P, proc Q}.

proc (out C V) -o {msg C V}.

proc (in C (x \ P x)) -o {pi (V \ msg C V -o {proc (P V)})}.

proc (rin C (x \ P x)) -o {!pi (V \ msg C V -o {proc (P V)})}.

proc (new (x \ P x)) -o {sigma (c \ proc (P c))}.

proc zero -o {one}.
```

□ Pi-calculus example

$$a(u).\mathtt{print}(u) \mid a(v).\mathtt{print}(a) \mid \bar{a}\langle b\rangle$$

Context starts with one process:

```
proc (par (in a (u \ print u)) (par (in a (v \ print a)) (out a b)))
```

Two uses of `par` rewriting rule produce:

```
proc (in a (u \ print u)), proc (in a (v \ print a)), proc (out a b)
```

`in` and `out` rewriting rules produce:

```
pi (V \ msg a V -o proc (print V)), pi (V \ msg a V -o proc (print a)), msg a
```

Finally, first process consumes `msg`:

```
proc (print b), pi (V \ msg a V -o proc (print a))
```

Note: other result also possible.

□ Graph Bipartiteness Checking

LolliMon version of Ganzinger-McAllester logical algorithm.

- Graph edges stored in unrestricted context.
  At start, assume edges already in context and nodes given as explicit input.

- Stores labelling information in unrestricted context, relies on saturation to finish label propagation.

- Deletion modeled by linear hypotheses– unlabeled nodes stored in linear context.

- Priorities modeled by monadically separated phases.

□ Graph Label Propagation

```
(* propagate label constraints *)
!labeled U a,
!edge U V
-o {!labeled V b}.

!labeled U b,
!edge U V
-o {!labeled V a}.

(* remove all nodes that have been labeled *)
!labeled U K,
unlabeled U
-o {one}.
```

□ Phases of Bipartite Checking

```
(* start program *)
notbipartite Us <= {init Us -o {iterate}}.

(* create symmetric closure *)
!edge U V -o {!edge V U}.

(* initialize nodes as unlabeled *)
init (U::Us) -o {unlabeled U, init Us}.
init nil -o {one}.

(* label propagation *)
iterate o- sigma U \ !labeled U a, !labeled U b, top.
iterate o- sigma U \ unlabeled U, (labeled U a => {iterate}).
```

□ Bipartite Checking Execution

Initial Context

$\Delta =$

$\Gamma =$ `edge n1 n2, edge n1 n3, edge n2 n3`

Symmetric Closure

$\Delta =$

$\Gamma =$ `edge n1 n2, edge n2 n1, edge n1 n3, edge n3 n1, edge n2 n3, edge n3 n2`

Initialize nodes

$\Delta =$ `init (n1::n2::n3),`

$\Gamma =$ `edge n1 n2, edge n2 n1, edge n1 n3, edge n3 n1, edge n2 n3, edge n3 n2`

$\vdots$

$\Delta =$ `unlabeled n1, unlabeled n2, unlabeled n3`

$\Gamma =$ `edge n1 n2, edge n2 n1, edge n1 n3, edge n3 n1, edge n2 n3, edge n3 n2`

□ More Bipartite Checking Execution

Label propagation

$\Delta =$ `unlabeled n2, unlabeled n3`

$\Gamma =$ `labeled n1 a,`

`edge n1 n2, edge n2 n1, edge n1 n3, edge n3 n1, edge n2 n3, edge n3 n2`

$\vdots$

$\Delta =$

$\Gamma =$ `labeled n1 a, labeled n1 b, labeled n2 a, labeled n2 b,`

`labeled n3 a, labeled n3 b,`

`edge n1 n2, edge n2 n1, edge n1 n3, edge n3 n1, edge n2 n3, edge n3 n2`

Saturation reached.

New propagation phase ends immediately since there exists a node with both labels.

Graph is not bipartite.

□ Other Examples

- Linear destination passing operational semantics.
- Other logical algorithms— parsing, union-find, etc.
- Meta-circular interpreter
- Simple Theorem Provers
- Petri-nets
- Lolli programs

□ Future Work

- More algorithms– e.g. ”linear logical” algorithms
- Better implementation techniques
- Integration of full-theorem prover or model checker

□ Related Work

- CLF [Watkins et. al.]
- Forum [Miller]
- Prolog-style linear logic programming—
  Lolli [Hodas & Miller], Lygon [Harland et. al.]
- Concurrent linear logic programming—
  LO [Andreoli & Pareschi], ACL [Kobayashi & Yonezawa]
- Forward and backward chaining in linear logic
  [Harland et. al.]
- Bottom-up linear logic programming [Bozzano et. al.]

□ Summary

LolliMon is a novel logic-programming language.

- Conservative extension of Lolli
- Integrates backward chaining and forward chaining.
- New (we think) use of monad in logic programming.

Prototype implementation at: `www.cs.cmu.edu/~fp/lollimon`