Distributed deductive databases, declaratively:
The L10 logic programming language

Robert J. Simmons
Bernardo Toninho
Frank Pfenning

Computer Science Department
Carnegie Mellon University

June 4, 2011
Summary

1. Introduction

2. L10 Language Features

3. On Implementing L10

4. Future Work and Conclusion
Introduction

What is L10?

- Forward-chaining logic programming language,
- Distribution of data and parallelism of computation,
- Logically motivated notion of worlds as locations for computation.

Forward-chaining logic programming

- Deals with collections of facts that model some structure;
- Computation is described by rules;
- Operational interpretation of rules is exhaustive forward deduction:
  - Try to match facts from the database against the premises of a rule;
  - Add the conclusion to the database, if not already present;
  - Computation terminates when no new facts can be added.
What is L10?

- *Forward-chaining* logic programming language,
- Distribution of data and parallelism of computation,
- Logically motivated notion of worlds as locations for computation.

Forward-chaining logic programming

- Deals with collections of facts that model some structure;
- Computation is described by rules;
- Operational interpretation of rules is exhaustive forward deduction:
  - Try to match facts from the database against the premises of a rule;
  - Add the conclusion to the database, if not already present;
  - Computation terminates when no new facts can be added.
Introduction
What is L10? - Forward-chaining

Transitive closure of a graph
We can encode a graph as a fact database:

\[
\begin{align*}
\text{edge } & \text{a c} \quad \text{edge } & \text{c a} \quad \text{edge } & \text{d c} \\
\text{edge } & \text{a d} \quad \text{edge } & \text{c d} \quad \text{edge } & \text{d d} \\
\text{edge } & \text{b b} \quad \text{edge } & \text{d b}
\end{align*}
\]

edge X Y -> path X Y.
edge X Y, path Y Z -> path X Z.

Why forward-chaining?
- A natural way to describe fixed-point iteration and database-like algorithms;
- Can produce surprisingly succinct and efficient programs.
Introduction
What is L10? - Forward-chaining

Transitive closure of a graph
We can encode a graph as a fact database:

```
edge a c  edge c a  edge d c
edge a d  edge c d  edge d d
edge b b  edge d b
```

edge X Y -> path X Y.
edge X Y, path Y Z -> path X Z.

Why forward-chaining?
- A natural way to describe fixed-point iteration and database-like algorithms;
- Can produce surprisingly succinct and efficient programs.
Introduction
What is L10? - Forward-chaining

Transitive closure of a graph

We can encode a graph as a fact database:

- edge a c
- edge c a
- edge d c
- edge a d
- edge c d
- edge d d
- edge b b
- edge d b

\[
\begin{align*}
\text{edge } X Y & \rightarrow \text{path } X Y. \\
\text{edge } X Y, \text{ path } Y Z & \rightarrow \text{path } X Z.
\end{align*}
\]

Why forward-chaining?

- A natural way to describe fixed-point iteration and database-like algorithms;
- Can produce surprisingly succinct and efficient programs.
1 Introduction

2 L10 Language Features

3 On Implementing L10

4 Future Work and Conclusion
Distributed programming in L10

The foundation of distributed programming in L10 is *worlds*:

- Abstractly represent different storage and computation locations;
- All relations in L10 must be associated with a declared world;
- Dependencies between relations result in dependencies between worlds.

Example: Liveness analysis declaration

L10 is a typed language. All worlds and relations must be declared:

```plaintext
wLive : world.
live : nat -> t -> rel @ wLive.
```
Distributed programming in L10

The foundation of distributed programming in L10 is *worlds*:

- Abstractly represent different storage and computation locations;
- All relations in L10 must be associated with a declared world;
- Dependencies between relations result in dependencies between worlds.

Example: Liveness analysis declaration

L10 is a typed language. All worlds and relations must be declared:

```plaintext
wLive : world.
live : nat -> t -> rel @ wLive.
```
L10 Language Features

Constructive Negation

L10 worlds stage computation by determining the order in which relations are computed.

Program Analyses - Liveness

Information about the program code is encoded using these relations:

- \text{def } L X - X \text{ is defined in line } L.
- \text{use } L X - X \text{ is used in line } L.
- \text{succ } L L' - L' \text{ may be executed immediately after } L.

Liveness is defined by the two rules:

\text{use } L X \rightarrow \text{ live } L X.

\text{live } L' X, \text{ succ } L L', \text{ not } (\text{def } L X) \rightarrow \text{ live } L X.

Negation in forward-chaining logic programming can be problematic...
Some uses of negation can make sense. For instance:

\[
\text{not (fact2)} \rightarrow \text{fact1}
\]

where we can stage computation such that fact2 is completely determined when we are considering this rule for fact1.

wCode : world.
def : nat -> t -> rel @ wCode.
...
live : nat -> t -> rel @ wLive.
use L X -> live L X.
live L' X, succ L L', not (def L X) -> live L X.

No cyclic dependencies between worlds are allowed in L10.
Some uses of negation can make sense. For instance:

$$\neg (\text{fact2}) \rightarrow \text{fact1}$$

where we can stage computation such that \text{fact2} is \textit{completely} determined when we are considering this rule for \text{fact1}.

wCode : world.
def : nat -> t -> rel @ wCode.
...
live : nat -> t -> rel @ wLive.
use L X -> live L X.
live L' X, succ L L', not (def L X) -> live L X.

No \textit{cyclic} dependencies between worlds are allowed in L10.
Exploiting worlds for parallelism

We can safely stage independent worlds for parallel execution.

Program Analyses - Neededness

We can define a neededness analysis:

- \( \text{nec} \ L \ X \ @ \ w\text{Code} \): at line \( L \), \( X \) is necessary for control flow or as the return value.
- \( \text{needed} \ L \ X \ @ \ w\text{Need} \): at line \( L \), \( X \) is needed.

\[
\text{nec} \ L \ X \rightarrow \text{needed} \ L \ X.
\]

\[
\text{needed} \ L' \ X, \text{succ} \ L \ L', \not (\text{def} \ L \ X) \rightarrow \text{needed} \ L \ X.
\]

\[
\text{use} \ L \ Y, \text{def} \ L \ X, \text{succ} \ L \ L', \text{needed} \ L' \ X \rightarrow \text{needed} \ L \ Y.
\]

This way, the liveness and neededness analyses can be executed in parallel.
Exploiting worlds for parallelism

We can safely stage independent worlds for parallel execution.

Program Analyses - Neededness

We can define a neededness analysis:

- nec L X @ wCode: at line L, X is necessary for control flow or as the return value.
- needed L X @ wNeed: at line L, X is needed.

\[
\begin{align*}
nec L X & \to needed L X. \\
needed L' X, succ L L', not (def L X) & \to needed L X. \\
use L Y, def L X, succ L L', needed L' X & \to needed L Y.
\end{align*}
\]

This way, the liveness and neededness analyses can be executed in parallel.
Suppose we want to implement a regular expression matcher. The type `regexp` captures the form of reg. expressions. Tokens will be represented by string constants.

### Reg. exp. matcher declaration

```plaintext
regexp : type.
tok : string -> regexp.
emp : regexp.
altp : regexp -> regexp -> regexp.
negp : regexp -> regexp.
```

```plaintext
w0 : world.
w1 : regexp -> world.
token : string -> nat -> rel @ w0.
match : {RE : regexp} nat -> nat -> rel @ w1 RE.
```
Suppose we want to implement a regular expression matcher. The type `regexp` captures the form of reg. expressions. Tokens will be represented by string constants.

Reg. exp. matcher declaration

```plaintext
regexp : type.
tok : string -> regexp.
emp : regexp.
alt : regexp -> regexp -> regexp.
neg : regexp -> regexp.

w0 : world.
w1 : regexp -> world.
token : string -> nat -> rel @ w0.
match : {RE : regexp} nat -> nat -> rel @ w1 RE.
```
Indexed worlds and limited saturation

... token _ I -> match emp I I.

match RE1 I J -> match (alt RE1 RE2) I J.
match RE2 I J -> match (alt RE1 RE2) I J.

Most deductive databases would not allow these rules for alternation!

token _ I, token _ J, I <= J,
not (match RE I J) -> match (neg RE) I J.

Negation is justified by locally stratified negation.
... 

token _ I -> match emp I I.

match RE1 I J -> match (alt RE1 RE2) I J.
match RE2 I J -> match (alt RE1 RE2) I J.

Most deductive databases would not allow these rules for alternation!

token _ I, token _ J, I <= J,
not (match RE I J) -> match (neg RE) I J.

Negation is justified by \textit{locally stratified negation}. 
Indexed worlds and limited saturation

... token _ I -> match emp I I.

match RE1 I J -> match (alt RE1 RE2) I J.
match RE2 I J -> match (alt RE1 RE2) I J.

Most deductive databases would not allow these rules for alternation!

token _ I, token _ J, I <= J,
not (match RE I J) -> match (neg RE) I J.

Negation is justified by *locally stratified negation*. 
1 Introduction

2 L10 Language Features

3 On Implementing L10

4 Future Work and Conclusion
On Implementing L10
Scheduling

Static scheduling

How to handle queries for non-indexed worlds:

1. Compute the world dependency graph.
2. Perform a breadth-first traversal of the graph.
3. Produce a task list that maps L10 worlds to available X10 places.

Scheduling program analyses

Assuming two X10 places, A and B:

1. Worlds \( w_{\text{Live}} \) and \( w_{\text{Need}} \) depend on world \( w_{\text{Code}} \), which depends on no worlds.
2. BFS traversal: \( w_{\text{Code}} \), followed by \( w_{\text{Live}} \) and \( w_{\text{Need}} \).
3. Schedule \( w_{\text{Code}} \) and \( w_{\text{Live}} \) at place A, \( w_{\text{Need}} \) at place B. Computation at place B blocks until relations at \( w_{\text{Code}} \) are computed.
On Implementing L10

Scheduling

Static scheduling

How to handle queries for non-indexed worlds:

1. Compute the world dependency graph.
2. Perform a breadth-first traversal of the graph.
3. Produce a task list that maps L10 worlds to available X10 places.

Scheduling program analyses

Assuming two X10 places, A and B:

1. Worlds wLive and wNeed depend on world wCode, which depends on no worlds.
2. BFS traversal: wCode, followed by wLive and wNeed.
3. Schedule wCode and wLive at place A, wNeed at place B. Computation at place B blocks until relations at wCode are computed.
 Indexed worlds are more interesting:

- Perform a BF traversal of the relevant subterm indices of the world:
  - until all subterms have been considered,
  - or the number of unique branches exceeds available parallelism.

Scheduling a regular expression match

Assuming 3 X10 places A, B and C, a matching for $\neg(b | c)(a^+)$ is scheduled as:

```
Place C  w1(b)

Place B  w1(c) ← w1(b | c) ← w1(\neg(b | c))

Place A  w0 ← w1(a) ← w1(a+) ← w1(\neg(b | c)(a+))
```
Integration with X10

We do not specify how to query saturated databases:

- Such queries will be performed through an API within X10
- Main uses of L10 programs are to provide data to other programs
- The language will eventually be available through an X10 library
- Similar APIs exist for many deductive database/programming language combinations

L10 is still at a very early development stage:

- Fully functional interpreter (written in Standard ML) - fully sequential.
- Compiler to Standard ML and X10 - in development.
Summary

1. Introduction
2. L10 Language Features
3. On Implementing L10
4. Future Work and Conclusion
Future Work & Conclusions

Future Work

- Indexing worlds with non-structured terms (e.g. strings, numbers)
- Optimizing communication between worlds/places
- Finishing the implementation of the compiler...

Conclusions

- Introduced the preliminary design of a rich distributed logic programming language
- Exploit a mapping of the logically motivated notion of worlds to X10 places,
- Take advantage of inherent parallelism present in logic programs
Future Work & Conclusions

Future Work

- Indexing worlds with non-structured terms (e.g. strings, numbers)
- Optimizing communication between worlds/places
- Finishing the implementation of the compiler...

Conclusions

- Introduced the preliminary design of a rich distributed logic programming language
- Exploit a mapping of the logically motivated notion of worlds to X10 places,
- Take advantage of inherent parallelism present in logic programs
Suggestions are welcomed...