Distributed deductive databases, declaratively: The L10 logic programming language

Robert J. Simmons
Bernardo Toninho
Frank Pfenning

Computer Science Department
Carneggie Mellon University

June 4, 2011

Summary

1 Introduction

2 L10 Language Features

3 On Implementing L10

4 Future Work and Conclusion

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

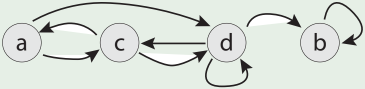
- Deals with collections of facts that model some structure;
- Computation is described by rules;
- Operational interpretation of rules is exhaustive forward deduction:
  - Try to match facts from the database against the premises of a rule;
  - Add the conclusion to the database, if not already present;
  - 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:



$$\left\{\begin{array}{lll} \texttt{edge a c} & \texttt{edge c a} & \texttt{edge d c} \\ \texttt{edge a d} & \texttt{edge c d} & \texttt{edge d d} \\ \texttt{edge b b} & \texttt{edge d b} & \end{array}\right\}$$

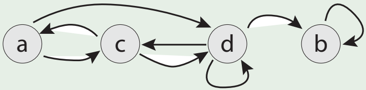
```
edge X Y -> path X Y.
edge X Y, path Y Z -> path X Z.
```

Why forward-chaining?

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

Transitive closure of a graph

We can encode a graph as a fact database:



$$\left\{\begin{matrix} \texttt{edge a c} & \texttt{edge c a} & \texttt{edge d c} \\ \texttt{edge a d} & \texttt{edge c d} & \texttt{edge d d} \\ \texttt{edge b b} & \texttt{edge d b} & \end{matrix}\right\}$$

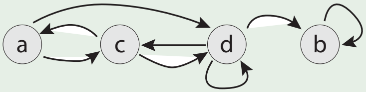
```
edge X Y -> path X Y.
edge X Y, path Y Z -> path X Z.
```

Why forward-chaining?

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

Transitive closure of a graph

We can encode a graph as a fact database:



$$\left\{\begin{array}{lll} \texttt{edge a c} & \texttt{edge c a} & \texttt{edge d c} \\ \texttt{edge a d} & \texttt{edge c d} & \texttt{edge d d} \\ \texttt{edge b b} & \texttt{edge d b} & \end{array}\right\}$$

```
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 succint and efficient programs.

Summary

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:

```
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:

```
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:

- `def L X` - `X` is defined in line `L`.
- `use L X` - `X` is used in line `L`.
- `succ L L'` - `L'` may be executed immediately after `L`.

Liveness is defined by the two rules:

```
use L X -> live L X.

live L' X, succ L L', not (def L X) -> live L X.
```

Negation in forward-chaining logic programming can be problematic. . .

Stratified Negation

Some uses of negation can make sense. For instance:

```
not (fact2) -> 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.

Stratified Negation

Some uses of negation can make sense. For instance:

```
not (fact2) -> 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.

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.

```
nec L X -> needed L X.
needed L' X, succ L L', not (def L X) -> needed L X.

use L Y, def L X, succ L L', needed L' X -> 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.

```
nec L X -> needed L X.
needed L' X, succ L L', not (def L X) -> needed L X.

use L Y, def L X, succ L L', needed L' X -> needed L Y.
```

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

```
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.
```

L10 Language Features

Indexed worlds and limited saturation

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

```
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.
```

```
...
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 *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 *locally stratified negation*.

Summary

1 Introduction

2 L10 Language Features

3 On Implementing L10

4 Future Work and Conclusion

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.

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.

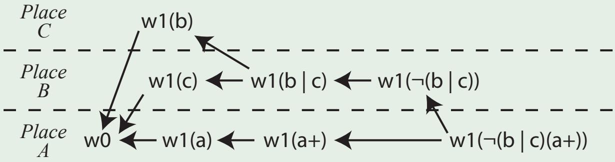
Scheduling indexed worlds

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 $\neg(b \mid c)(a+)$ is scheduled as:



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...