Distributed deductive databases, declaratively

The L10 logic programming language

Robert J. Simmons   Bernardo Toninho   Frank Pfenning
Carnegie Mellon University
{rjsimmon,btoninho,fp}@cs.cmu.edu

Abstract

We present the design of **L10**, a rich forward-chaining (a.k.a. "bottom-up") logic programming language. **L10** allows parallel computation to be explicitly specified through the use of *worlds*, a logically-motivated concept that has been used to describe distributed functional programming. An interpreter for **L10** runs these logic programs on top of the infrastructure of the **X10** programming language, and is responsible for mapping between **L10**'s worlds and *places*, the related **X10** construct for describing distributed computation.

Categories and Subject Descriptors D.1.6 [*Programming Techniques*]: Logic programming

General Terms Design, Languages

Keywords distributed programming, logic programming, X10

1. Introduction

Forward-chaining logic programming is a way to declaratively specify many algorithms, particularly those that involve database-like operations or iteration to a fixed point, in a succinct and natural way. In this paper, we present the design and preliminary implementation of the **L10** language, which permits explicit declarations of parallelism through the use of *worlds*. Worlds are similar to **X10**'s *places* in that they represent potentially discrete locations where data can be stored and computation can take place. However, the concepts are not identical, and one role of an implementation of **L10** is to map worlds onto places in such a way that the maximum amount of useful parallelism can be exposed.

1.1 Forward-chaining logic programming

Forward-chaining logic programming deals with a collections of facts (which we think of as a finite relation) that model some structure. Computations on these structures are described by *rules*. For instance, the following two rules specify a `path` relation as the transitive closure of a relation `edge` that represents edges in a graph. The first rule says that any edge is also a path, and the second rule says that we can extend every path along an edge.

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

X10 '11 June 4, 2011, San Jose, California
Copyright © 2011 ACM SIGPLAN. . . $10.00

In the first rule, `edge X Y` is the *premise* and `path X Y` is the *conclusion*. We use a syntax for rules which emphasizes that rules are just logical implications; a reader familiar with Prolog notation would expect to see the second rule written as "`path(X,Z) :- edge(X,Y), path(Y,Z)`".

The operational interpretation of these rules is *exhaustive forward deduction*. This means that we repeatedly take facts from our database and try to match them against the premises of our rules; whenever we succeed, we add the conclusion of that rule to the database if that fact is not already present. Once no new facts can be derived, we say that the database is *saturated* and forward deduction terminates. Systems implementing forward-chaining logic programming are often called *deductive databases*, as they perform exhaustive forward deduction over databases of facts to compute other databases of facts.

Forward-chaining logic programming is a natural way of specifying many important algorithms. Two particularly important papers in this area are Shieber, Schabes, and Pereira's work on specifying parsing algorithms as forward-chaining logic programs [10] and McAllester's work on specifying program analyses [6]. McAllester's work, which also showed that a suitable interpreter permits high-level reasoning about the asymptotic time complexity of logic programs, has been particularly influential.

In addition to McAllester's foundational theoretical work, recent work has shown that large-scale program analysis is possible using simple forward-chaining rules on top of efficient Datalog implementations [1, 2, 4, 5, 15, 16]. The experience of the BDDB-DDB project in particular was that the logical specification of Java pointer analysis, in addition to being orders of magnitude more concise than hand-tuned analyses written in Java, could be executed twice as fast as those hand-coded analyses [15].

1.2 Distributed programming with worlds

The foundation of distributed programming in **L10** is the *world*. Worlds abstractly represent (potentially) different locations for the storage and computation of relations. All relations must be explicitly declared in **L10** programs, and the declaration of a relation must associate it with some declared world.

We will use as an example a *liveness analysis* on programs written in a simple low-level language. The result of liveness analysis is a two place relation `live` between line numbers in the program and variable names – if `live L X` is in the database, then when the program counter is set to `L` the contents of variable `X` cannot be discarded. We declare this relation in **L10** as follows:

```
wLive : world.
live : nat -> t -> rel @ wLive.
```

The first declaration introduces a single world, `wLive`, and the second declaration introduces a two-place relation `live` that exists at world `wLive` (the keyword `rel` stands for "relation"). Line numbers are represented by the built-in type `nat` of natural numbers, and variables are represented by the built-in type `t` of arbitrary constants.

Liveness analysis depends on the program's code, and there may be other analyses (such as neededness analysis) that also depend on the program's code but that are independent of liveness analysis. If we put the relations that encode the program and the relations that encode neededness analysis at separate worlds (say, `wCode` and `wNeed`), it allows the **L10** interpreter to potentially map these worlds to different **X10** places. If `wCode` and `wLive` are mapped to different **X10** places, then computing the `live` relation will involve communication between **X10** places. If `wLive` and `wNeed` are mapped to different **X10** places, then the computation of liveness and neededness analysis can happen in parallel.

The idea of parameterizing relations by worlds is not an arbitrary choice; it has a logical basis in the intuitionistic Kripke semantics for modal logics as explored by Simpson [13]. Murphy has previously shown that Simpson's explicit worlds can be used as the basis for a distributed programming language [7, 8]. Murphy's language, **ML5**, is a ML-like language for distributed web programming that is in some ways similar to **X10**. In both **ML5** and **L10** different worlds allow data to exist in different physical locations, but **ML5** allows back-and-forth communication between worlds whereas the communication in **L10** is necessarily one-way.

1.3 Constructive negation with worlds

Worlds have another important use in **L10**. They *stage* the computation by determining the order in which relations are computed. Consider the world `wCode` mentioned above. Information about the program's code can be encoded in three relations that exist at world `wCode` – `succ L L'` denotes that we may execute line `L'` immediately after executing line `L`, `def L X` denotes that `X` is defined at line `L`, and `use L X` denotes that `X` is used at line `L`. (We show how these relations are computed in Section 2.1.1.) Given these relations, liveness analysis can be defined by two rules. First, a variable is live on any line where it is used:

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

Second, if a variable `X` is live at line `L'`, it is live at all the predecessors of `L'` that do not, themselves, define `X`:

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

The second rule must be treated carefully, because it depends on a fact about the `def` relation *not* holding. Negation introduces a well-known complication to the execution of logic programs: a rule such as "`not fact -> fact`" can cause inconsistent behavior in a logic programming interpreter that checks premises ("Is `fact` in the database? No.") and then asserts conclusions ("Okay, then add `fact` to the database.") The theory of *stratified negation* argues that some uses of negation make sense. If we have a rule "`not fact1 -> fact2`" and if it is possible to stage the computation to ensure that, when this rule is considered, there can be no additional facts about `fact1`, then we are justified in applying the rule and deriving `fact2`. In **L10**, we use worlds to stage computation, so the second rule asserts that we have to do all necessary computation at world `wCode` *first*, before we try to do any computation at `wLive`. It must also be the case that `wCode` cannot depend on `wLive`, or else we would not be able to stage the computation appropriately. **L10** programs forbid any cyclic dependencies between worlds for this reason.

While deductive databases have long allowed for stratified negation of various kinds, they have always justified negation in terms of models that assign Boolean truth values to facts. Research into *constructive provability logic* provides a proof-theoretic justification for **L10**'s implementation of staging and stratified negation [11, 12]. The details of the exact relationship between **L10** and constructive provability logic are outside the scope of this paper, however.

1.4 Summary

L10 is a forward-chaining logic programming language that uses a logically-motivated notion of *worlds* for two different purposes: the explicit declaration of parallelism (Section 1.2) and program staging, which enables stratified negation (Section 1.3).

Using the same logical mechanism for these two purposes, even though they are somewhat related, does introduce some tension into our language. As an example, we might really want the computation and data for both `wCode` and `wLive` to take place at the same **X10** place, but we are forced to use two different worlds, which may be mapped to different places, in order to refer negatively to the `def` relation when defining the `live` relation.

In Section 2, we will discuss a few more aspects of the **L10** language through a series of examples. In Section 3 we will discuss how Elton, the prototype interpreter for the **L10** language, operates. The Elton implementation is available from `http://l10.hyperkind.org`: a sequential interpeter exists, and the parallel interpreter is still in development. In Section 4 we conclude and discuss some future work.

2. Features of the L10 language

In the previous section, we gave an overview of the primary high-level features of **L10**: exhaustive forward deduction and explicit worlds for specifying parallelism and staging computation. In this section, we will give several more examples that go into more detail about the features and expressiveness of our language.

2.1 Parallel program analyses

Much recent interest in forward-chaining logic programming has come from the compiler and program analysis communities; many important program analyses can be given very concise and natural specifications, as well as efficient implementations, through the use of deductive databases [1, 4, 6, 15]. In this section, we consider a small low-level intermediate language in a compiler with three-address operations. The goal is to specify liveness and neededness analysis in logical form, and we will see that the natural specifications exhibit some parallelism that can be exploited. As discussed in the introduction, **L10**'s worlds are used both to enable stratified negation and to expose this natural parallelism.

For the purpose of the example, our language has the instructions shown below. We use x, y, z for variables, c for constants, $\oplus$ (*op*) for binary operations, and ? (*cmp*) for comparison operations. We use l for line numbers, which are represented as natural numbers (of type `nat`) in **L10**; comparisons and addition for natural numbers are primitives in the language.

The informal notation for these analyses, taken from Pfenning's lecture notes for a Compiler Design course (available from `http://www.cs.cmu.edu/~fp/courses/15411-f09/`), is given below on the left; the encoding of these facts in **L10** is given on the right.

$l : x \leftarrow y \oplus z$	`line L (binop X Y Op Z)`
$l : x \leftarrow y$	`line L (move X Y)`
$l : x \leftarrow c$	`line L (loadc X C)`
l : goto l'	`line L (goto L')`
l : if $(x \; ? \; c)$ goto l'	`line L (if X Cmp C L')`
l : return x	`line L (return X)`

We capture instructions as a type `inst`, declared on line 6 in Figure 1. **L10** allows user-defined types in addition to the three

```
// Commands

inst: type.
binop:  t -> t -> t -> t -> inst.
move:   t -> t -> inst.
loadc:  t -> t -> inst.
goto:   nat -> inst.
if:     t -> t -> t -> nat -> inst.
return: t -> inst.

wCode: world.
line: nat -> inst -> rel @ wCode.

// Extracting relevant information

succ: nat -> nat -> rel @ wCode.
def:  nat -> t -> rel @ wCode.
use:  nat -> t -> rel @ wCode.

line L (binop X Y Op Z) ->
   succ L (L+1),
   def L X,
   use L Y, use L Z.

line L (move X Y) ->
   succ L (L+1),
   def L X,
   use L Y.

line L (loadc X C) ->
   succ L (L+1),
   def L X.

line L (goto L') ->
   succ L L'.

line L (if X Cmp C1 L') ->
   succ L L', succ L (L+1),
   use L X.

line L (return X) ->
   use L X.
```

Figure 1. Program analysis: capturing program information.

built-in types `string` (string literals), `nat` (nonnegative integers), and `t` (an open-ended type of arbitrary constants).

2.1.1 Extracting program information

The first phase of the analysis extracts relevant information from the program, which is represented as facts of the form above. Both the description of the program and the extracted information are stored at the **L10** world `wCode`. There are three relevant relations here, at least initially:

- $\mathsf{succ}\ l\ l'$: line l has (potential) successor l' in the program CFG.
- $\mathsf{def}\ l\ x$: line l *defines* variable x.
- $\mathsf{use}\ l\ x$: line l *uses* variable x.

Given a line of code, this first stage in the analysis extracts the successors, defined variables, and used variables. For instance, a binary operation has one successor, defines one variable, and uses two variables, whereas a conditional jump has two (potential) successors, defines no variables, and uses one variable. These two

```
nec: nat -> t -> rel @ wCode.
line L (if X Comp C L1) -> nec L X.
line L (return X) -> nec L X.

wNeed: world.
needed: nat -> t -> rel @ wNeed.

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.
```

Figure 2. Program analysis: neededness.

rules are logically represented in informal notation as follows:

$$\frac{l : x \leftarrow y \oplus z}{\begin{array}{l}\mathsf{succ}\ l\ (l+1)\\ \mathsf{def}\ l\ x\\ \mathsf{use}\ l\ y\\ \mathsf{use}\ l\ z\end{array}}\,J_1 \qquad \frac{l : \mathsf{if}\ (x\ ?\ c)\ \mathsf{goto}\ l'}{\begin{array}{l}\mathsf{succ}\ l\ (l+1)\\ \mathsf{succ}\ l\ l'\\ \mathsf{use}\ l\ x\end{array}}\,J_5$$

The **L10** code for this portion of the analysis can be seen in Figure 1. The language allows rules to have multiple conclusions, though all the relations in a conclusion must be defined at the same world.

2.1.2 Liveness analysis

With `succ`, `def`, and `use` defined, we now can implement liveness analysis as described in Section 1. Usually, liveness analysis is presented in the form of data flow equations for which we compute a least fixed point. Here, however, we simply run the two rules from Section 1.3 to saturation, which can also be seen as a least fixed point computation.[1] The second rule presented there makes it clear that liveness uses a form of backward propagation: from the knowledge that x is live at l' we infer that x is live at l under certain conditions.

2.1.3 Neededness analysis

The **L10** program we have been developing has used worlds for stratified negation, but we have not yet explored any opportunities for parallelism. We will now consider a neededness analysis that can inform dead-code elimination and run in parallel to liveness analysis. The liveness information computed in the previous section is not appropriate for dead-code elimination, because an assignment such as $l : z \leftarrow z + x$ in a loop for a variable z which is not otherwise used will flag z as live throughout the loop, even though l is dead code. Slightly more precise is *neededness*. We will define two new relations:

- $\mathsf{nec}\ l\ x$: at line l, x is necessary for control flow or as the return value
- $\mathsf{needed}\ l\ x$: at line l, x is needed

[1] It is the least fixed point of the operator which extends the database of facts by all facts arising from executing all applicable rules.

```
wDead: world.
dead: nat -> rel @ wDead.

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

Figure 3. Program analysis: dead code.

The first relation is defined at world `wCode` like the `def`, `succ`, and use relations. The second relation is defined at world `wNeed`, seeded by these necessary variables and propagated backwards, similar to liveness analysis. The **L10** code for neededness analysis is given in Figure 2.

2.1.4 Dead-code elimination

Having performed a neededness analysis, identifying dead code (shown in Figure 3), is straightforward: code is dead if it defines a variable that is not needed. We introduce a relation `dead L`, meaning that the command at line `L` is dead code.

Since neededness and liveness analysis as presented above are independent, we can compute liveness in parallel with neededness and dead-code identification.

2.2 Regular expressions

Our next example is a regular expression matcher, which we will primarily use to introduce a new concept: worlds *indexed by first-order terms*. The type `regexp` captures the form of regular expressions over an arbitrary alphabet. Tokens will be represented by string constants.

Match the token a:	a	`tok "a"`
Match the empty string:	ϵ	`emp`
Match r once or more:	$r+$	`some RE`
Match r_1 and r_2 in sequence:	$r_1 r_2$	`seq RE1 RE2`
Match either r_1 or r_2:	$r_1 \mid r_2$	`alt RE1 RE2`

Other common regular expressions can be defined with these primitives; for example, $r? \equiv (r \mid \epsilon)$ and $r* \equiv (\epsilon \mid r+)$.

Having described regular expressions, we can describe a regular expression matcher. We will use two relations. The string we are trying to match against the regular expression will be represented by the set of facts in the `token` relation, and we will introduce a three-place relation `match` which takes a regular expression and two positions, represented as natural numbers.

The fundamental difference between this example and those we have seen in the previous sections is that the `match` relation is associated with a world *indexed* by the matched regular expression. The declaration of the indexed world `w1` on line 13 of Figure 4 actually defines a family of worlds `w1(RE)`. When the head of a rule is a relation associated with world `w1(RE)` for some specific `RE`, the premises can refer to a relation associated with world `w1(RE')` if `RE'` is a *subterm* of `RE`. For instance, relations associated with the world `w1(alt emp (tok "a"))` can depend on relations associated with worlds `w1(emp)` and `w1(tok "a")`, but not on relations associated with the world `w(tok "b")`. As we will later see, this is crucial to ensure termination of our matcher.

The declaration of the `match` relation has to specify the relationship between the arguments of the relation and the world's index. We do this by assigning a name, `RE`, to the first argument when we declare the `match` relation on line 15 of Figure 4. The notation "`{RE: regexp} nat ->...`" is equivalent to the notation "`regexp -> nat ->...`" that we have been using, but it allows the argument to be named and mentioned later on in the declaration. Names can always be provided, so we could also have written "`{RE: regexp} {I: nat}...`" if we wanted.

```
1   // Regular expressions
2
3   regexp : type.
4   tok:  string -> regexp.
5   emp:  regexp.
6   some: regexp -> regexp.
7   seq:  regexp -> regexp -> regexp.
8   alt:  regexp -> regexp -> regexp.
9
10  // Parsing regular expressions
11
12  w0: world.
13  w1: regexp -> world.
14  token: string -> nat -> rel @ w0.
15  match: {RE: regexp} nat -> nat -> rel @ w1 RE.
16
17  token T I -> match (tok T) I (I+1).
18
19  token _ I -> match emp I I.
20
21  match RE I J -> match (some RE) I J.
22
23  match RE I J,
24  match (some RE) J K ->
25     match (some RE) I K.
26
27  match RE1 I J,
28  match RE2 J K ->
29     match (seq RE1 RE2) I K.
30
31  match RE1 I J -> match (alt RE1 RE2) I J.
32
33  match RE2 I J -> match (alt RE1 RE2) I J.
```

Figure 4. Regular expression matching.

The rules that define the `match` relation, lines 17-33 in Figure 4, give the meaning of each regular expression constructor in a fairly straightforward manner. We can match a token if it occurs in a database (line 17); given an arbitrary token in a position i, we can always match the empty string in that position (line 19); if the string from i to j matches r_1 and the string from j to k matches r_2, then the string from i to k matches $r_1 r_2$ (lines 27-29).

2.2.1 Testing regular expressions

We encode a string as a series of facts, so the string "foo" is represented as this database:

$$\left\{ \begin{array}{ll} \texttt{token "f" 0} & \texttt{token "o" 2} \\ \texttt{token "o" 1} & \texttt{token "EOF" 3} \end{array} \right\}$$

and the string "boo" is represented as this database:

$$\left\{ \begin{array}{ll} \texttt{token "b" 0} & \texttt{token "o" 2} \\ \texttt{token "o" 1} & \texttt{token "EOF" 3} \end{array} \right\}$$

If our regular expression of interest is f(o+), then we can conclude that the string matches the regular expression if the fact `match (seq (tok "f") (some (tok "o"))) 0 3` is derivable from the database describing the string.

As mentioned in Section 1.1, the rules in a program are applied to known facts to derive new facts until no new information can be derived. We may then wonder how our regular expression matcher can reach saturation, considering that we can apparently always derive `match emp 0 0`, then `match (alt emp emp) 0 0`, and so on forever. Most deductive databases would not even allow a

```
1   // Adding negation to regular expressions
2
3   neg:  regexp -> regexp.
4
5   token _ I,
6   token _ J,
7   I <= J,
8   not (match RE I J) ->
9      match (neg RE) I J.
10
11  db3 = (token "d" 0, token "a" 1,
12         token "a" 2, token "EOF" 3)
13    @ w1 (seq (neg (alt (tok "b") (tok "c")))
14           (some (tok "a"))).
15
16  db4 = (token "b" 0, token "a" 1,
17         token "a" 2, token "EOF" 3)
18    @ w1 (seq (neg (alt (tok "b") (tok "c")))
19           (some (tok "a"))).
20
```

Figure 5. Regular expressions with negation, and two queries that try to match the strings "daa" (db3) and "baa" (db4) against the regular expression $\neg(b \mid c)(a+)$.

program like the one in Figure 4, because the rules dealing with alternation $(r_1 \mid r_2)$ on lines 31 and 33 of Figure 4 violate *range restriction*, a common requirement that all variables mentioned in a conclusion appear in a premise. This is a recurring pattern in forward-chaining logic programs, and a usual solution is to add a new relation, `subterm(RE)`, which enumerates the subterms of the regular expression we are interested in. Then, the rules on lines 31 and 33 of Figure 4 could be given the additional premise of `subterm(alt RE1 RE2)`, which would make the rules range restricted.

Adding an explicit `subterm` predicate is unnecessary in **L10**. When we made the regular expression argument an index to the world `w1`, it restricted us to writing programs where the derivability of a fact of the form `match RE I J` could only depend on the derivability of a fact of the form `match RE' I' J'` if `RE'` was a subterm of `RE`. Because we always know the form of the fact that we want to derive – in the motivating example, it was `match (seq (tok "f") (some (tok "o"))) 0 3` – then we can simply ask **L10** to only do the exhaustive forward-chaining necessary to prove this fact (if it is, in fact, provable). To this end, whenever we request that **L10** do exhaustive forward reasoning, we annotate the initial database with a world that limits how far saturation goes. This prevents the computation from diverging, since the problematic facts exist at worlds which are known to be irrelevant and so will never be considered.

To review, **L10** implements a notion of *limited saturation*: by annotating worlds with terms and requiring that facts at indexed worlds only depend on facts at the same world when the index is a subterm, we can capture a class of algorithms that naturally saturate *up to a point*. If an indexed world depends a non-indexed world – in the regular expression example, `w1` depends on `w0` – then all instances `w1(E)` of the indexed world depend on the non-indexed world. This feature serves the three purposes: it makes programs more concise by removing the need for extra `subterm` premises; it increases efficiency, since we only compute facts that exist in worlds that are relevant to the current computation; and it increases the number of opportunities for parallelism, since having worlds that depend on terms allows many more worlds to be independent from each other so that **L10** may perform the computation in parallel. This last point will be discussed further in Section 3.

2.2.2 Regular expressions with negation

As a final example, we will consider one extension to our regular expression program: the negation of a regular expression `neg(R)` (or, informally, $\neg r$). The rule, given on lines 5-9 of Figure 5, is straightforward – a string from i to j matches the regular expression $\neg r$ if it does not match the regular expression r.

While the intuitive meaning of the rule for negated regular expressions is clear, it is not immediately obvious that this use of negation is justified, as we are referring to the negation of the `match` relation to prove something about the `match` relation. It is justified since the world `w1` is indexed by a regular expression. The subterm ordering on regular expressions, which in the previous section allowed us to perform limited saturation, also ensures that we can stage computation at world `w1(RE)` before considering computation at world `w1(neg(RE))`. This use of stratified negation is one instance of *locally stratified negation*, which was first considered by Przymusinski [9].

3. Elton, the L10 interpreter

Elton is a prototype interpreter for the **L10** language; it is written in a combination of **X10** and Standard ML. Ultimately, we anticipate replacing most of the parts of Elton written in Standard ML with **L10**, so that the language is written entirely in **L10** and **X10**. The interpreter can be downloaded from `http://l10.hyperkind.org`.

Within a particular stage – that is, when performing computation at a particular world – the interpreter is not fundamentally different than a standard deductive database. Currently, this part of Elton uses inefficient data structures; we plan to implement indexed tuple-at-a-time evaluation that validates McAllester's cost semantics (at least when all evaluation is at a single **X10** place) [6].

3.1 Static scheduling

A unique aspect of Elton is that it enables parallelism by mapping different stages to different places in a coherent way. When a query is made, the interpreter will statically assign different stages to different **X10** places depending on the number of places that are available. When none of the worlds are indexed, this is done by making a breadth-first search of the world dependency graph. For example, if we wanted to compute both the liveness and dead code analyses of Section 2.1, then the computation would be scheduled on two different places as long as more than one place is available.

The result of this breadth-first search is a task list for every **X10** place. In the program analysis example, one possibility is that `wCode` and `wLive` will be scheduled at Place A and `wNeed` and `wDead` will be scheduled at Place B. In this case, computation at Place B will block until the the `def`, `succ`, `use`, and `nec` relations are derived at Place A.

3.2 Scheduling indexed worlds

The story is somewhat more interesting in the case when some of the worlds are indexed. In these cases, a breadth-first search of the (relevant) subterm indices of the world will be performed until either all subterms have been considered or the number of unique branches exceeds the available parallelism. As a concrete example, the regular expression queries from Figure 5 can be scheduled as shown in Figure 6 if there are at least three places available. This assignment is interesting in part because it is effectively the kind of search performed by a *backward-chaining* (a.k.a. "top-down") interpreter for logic programs in the style of Prolog.

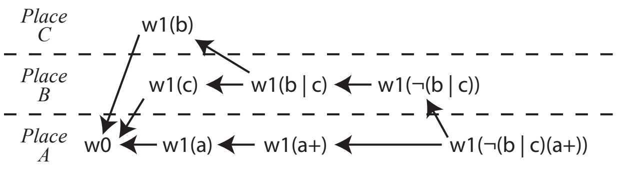

Figure 6. Place assignment for the queries in Figure 5

3.3 Integration with X10

We have specified a query syntax for triggering computations, but have not specified how the resulting saturated databases can be queried. One reason for this is that we expect such queries to be performed through an API within **X10**. While it is convenient to have a concrete syntax for specifying **L10** rules, many of the uses of **L10** logic programs are to provide data to functional or imperative programs (such as register allocation in the case of our alias analysis). Elton will eventually be accessible through an **X10** library that allows the programmer to load **L10** programs, specify databases, and query results. Similar APIs exist for many deductive database/programming language combinations; examples include Dyna, which exposes an API to C++ [3] and a McAllester-style interpreter that exposes an API to Standard ML [14].

4. Conclusion and future work

We have described the preliminary design and implementation of **L10**, a logic programming language that uses explicit worlds to stage computation and that uses the infrastructure of **X10** to take advantage of implicit parallelism in programs. There are many immediate opportunities to extend **L10** to add expressiveness, and there is also much to explore and evaluate in terms of efficiently executing **L10** programs in the context of **X10**. We will conclude by discussing some of this future work.

Minimizing communication Because it can be costly to transmit data from one place to another, it is important to be very clear about when non-local communication can take place. The current model for **L10** execution is that all necessary information is transmitted to the world associated with the conclusion(s) and dealt with there. It may be preferable in many situations to perform an indexed lookup at a different place and only communicate the result.

In unpublished work, Henry DeYoung has considered program transformations for epistemic logic programs that deal with these sorts of optimizations in a distributed setting, and applying his work to **L10** should allow us to automatically transform programs in a way that decreases communication costs.

Foundations in constructive provability logic The theoretical basis for **L10** is intended to be constructive provability logic [12], an intuitionistic modal logic that allows stratified negation to be modeled as regular intuitionistic negation. However, the theory of constructive provability logic is still lacking a few critical elements that must be addressed in order to ensure that **L10** as we have presented it here has a consistent logical basis. The first issue is that existing formalizations of constructive provability logic only capture propositional logic even though **L10** allows for first-order quantification. Similarly, the introduction of indexed worlds is unique to **L10** and is not modeled in current formalizations of constructive provability logic.

Distributed worlds In this paper, we have only considered using worlds indexed by structured terms where all the subterms could be obtained statically. Another way of distributing worlds indexed by strings or integers is by using a hash function to distribute relations over all available places. This would require a significant change to the static scheduling presented in Section 3, but the result would be the ability to describe MapReduce-style computations in **L10**.

Acknowledgments

Henry DeYoung contributed important insights to this paper. Support for this research was provided by an X10 Innovation Award from IBM, and by the Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program under Grants NGN-44 and SFRH / BD / 33763 / 2009.

References

[1] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '09)*, pages 243–261, 2009.

[2] M. Bravenboer and Y. Smaragdakis. Exception analysis and points-to analysis: better together. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '09)*, pages 1–12, 2009.

[3] J. Eisner, E. Goldlust, and N. A. Smith. Compiling comp ling: Weighted dynamic programming and the Dyna language. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing (HLT-EMNLP)*, pages 281–290, 2005.

[4] M. S. Lam, J. Whaley, B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the Symposium on Principles of Database Systems (PADS '05)*, 2005.

[5] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for Java. In *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*. Springer LNCS 3780, 2005.

[6] D. A. McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, 2002.

[7] T. Murphy VII. *Modal Types for Mobile Code*. PhD thesis, Carnegie Mellon University, 2008. Available as technical report CMU-CS-08-126.

[8] T. Murphy VII, K. Crary, R. Harper, and F. Pfenning. A symmetric modal lambda calculus for distributed computing. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science*, pages 286–295, 2004.

[9] T. C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In *Foundations of deductive databases and logic programming*, pages 193–216. Morgan Kaufmann Publishers Inc., 1988.

[10] S. M. Shieber, Y. Schabes, and F. C. N. Pereira. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1–2):3–36, 1995.

[11] R. J. Simmons and B. Toninho. Principles of constructive provability logic. Technical Report CMU-CS-10-151, School of Computer Science, Carnegie Mellon University, 2010.

[12] R. J. Simmons and B. Toninho. Constructive provability logic, 2011. Submitted, available from `http://l10.hyperkind.org`.

[13] A. K. Simpson. *The Proof Theory and Semantics of Intuitionistic Modal Logic*. PhD thesis, University of Edinburgh, 1994.

[14] J. M. Uecker. A library for bottom-up logic programming in a functional language. Bachelor's thesis, Jacobs University Bremen, 2010.

[15] J. Whaley. *Context-Sensitive Pointer Analysis using Binary Decision Diagrams*. PhD thesis, Stanford University, 2007.

[16] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using Datalog and binary decision diagrams for program analysis. In *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*. Springer LNCS 3780, 2005.