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 preliminary 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 preliminary design and implementation of the **L10** language, which permits explicit declarations of parallelism through the use of *worlds*. Worlds do not map precisely onto **X10**'s *places*, and one role of the **L10** implementation 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 begins with a set of facts that represent some structure. For example, this set (or database) of facts:

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

can be used to represent this directed graph:
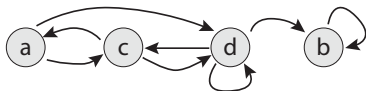
Similarly, this set of facts:

$$\left\{ \begin{array}{lll} \texttt{token "x" 1} & \texttt{token "y" 3} & \texttt{token "z" 5} \\ \texttt{token "+" 2} & \texttt{token "*" 4} & \end{array} \right\}$$

can be used to represent the string `"x+y*z"`. A set of facts with a common predicate (like `edge` or `token`) is thought of as a relation, so we can say `edge` is a *two-place relation* because `edge` facts always have two arguments.

Having represented structures as sets of facts, we specify computation by writing *rules*. For instance, the following two rules specify that the `path` relation is the transitive closure of the `edge` relation. 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.
```

In the first rule, `edge X Y` is the *premise* and `path X Y` is the *conclusion*. The second rule has two premises (`edge X Y` and `path Y Z`) and one conclusion (`path X Z`). 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 just means that we repeatedly take facts from our database and try to match them against the premises of our rules; whenever we succeed, we can 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. Given the example database above, this means that we could derive the fact `path d b` using the first rule and the fact `edge d b`. Using this newly-derived `path` fact, we could then use the second rule and the fact `edge a d` to derive `path a b`. Systems implementing forward-chaining logic programming are often called *deductive databases*, as they perform exhaustive forward deduction over databases of facts (like the `edge` relation) to compute other databases of facts (like the `path` relation).

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 [13] and McAllester's work on specifying program analyses [9]. McAllester's work, which also showed that a suitable interpreter permits high-level reasoning about the asymptotic time complexity of programs (later extended to space complexity by Liu et al. [7]), has been particularly influential.

In addition to McAllester's foundational theoretical work, a pair of recent projects have shown that large-scale program analysis for the Java language is possible using simple forward-chaining rules on top of BDD-based interpreters for forward-chaining logic

programs (in the case of the BDDBDDB project [6, 8, 18, 19]) or other highly-tuned Datalog implementations (in the case of the DOOP program analysis framework [2, 3]). The experience of the BDDBDDB 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 [18].

1.2 Distributed programming with worlds

The foundation of distributed programming in **L10** is *worlds*, which abstractly represent (potentially) different locations for the storage and computation of relations. All relations (like `edge`, `path`, and `tok`) must be explicitly declared in **L10** programs, and the declaration of a relation must associate it with some declared world. For our simple transitive closure program, we could use the following declarations. The keyword `rel` stands for "relation," so this declaration introduces a single world, `w` and introduces both `edge` and `path` as two-place relations (taking arbitrary terms with type `t`) that exist at world `w`.

```
w: world.
edge: t -> t -> rel @ w.
path: t -> t -> rel @ w.
```

The fact that both `edge` and `path` are declared at the same world means that the data about both the `edge` and `path` relations will be mapped to the same **X10** place. We could, alternatively, put both relations at their own worlds with the following declarations:

```
wEdge: world.
wPath: world.
edge: t -> t -> rel @ wEdge.
path: t -> t -> rel @ wPath.
```

These declarations would allow the **L10** interpreter to potentially map the two relations to different **X10** places. This is probably not what we want in our simple example, as it means that both of the rules in our program would have to communicate between **X10** places in order to compute the `path` relation. However, this sort of communication can be desirable if it allows a single relation (like `edge`) to influence multiple computations that happen in parallel. We will see an example of this in Section 2.1.

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 [16]. Murphy has previously shown that Simpson's explicit worlds can be used as the basis for a distributed programming language [10, 11]. 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, for reasons discussed below.

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. In the example above where the `edge` relation is assigned to world `wEdge` and the `path` relation is assigned to world `wPath`, the rules set up a dependency between the two worlds – if we are going to handle computation "one world at a time," then we have to do any computation at world `wEdge` *first*, before we try to do any computation at `wPath`.

This staging has a well-known consequence in terms of the use of negation and aggregates in logic programming. Negation must be used carefully in 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 it is justified to apply the rule and derive `fact2`.

Consider the following extension of our previous example. The first rule forces the `edge` relation to be symmetric, and the final rule computes a relation, `noedge`, that is the difference between the `edge` and `path` relations.

```
w: world.
w2: world.
edge: t -> t -> rel @ w.
path: t -> t -> rel @ w.
noedge: t -> t -> rel @ w2.

edge X Y -> edge Y X.
edge X Y -> path X Y.
edge X Y, path Y Z -> path X Z.
path X Y, not(edge X Y) -> noedge X Y.
```

Because the one rule that lets us derive `noedge` facts refers to the `edge` relation negatively, it is critical that the relations be at different worlds, as this allows us to stage all computation pertaining to the `edge` relation before we start computing `noedge`. Furthermore, because the last rule establishes that computation at world `w2` depends on computation on world `w`, it must also be the case that world `w` (where `edge` is defined) does not depend on world `w2`. **L10** programs forbid any cyclic dependencies between worlds for this reason.

While deductive databases have long allowed for stratified negation of various kinds, they did not have a satisfying logical justification. Research into *constructive provability logic* provides the logical justification for **L10**'s implementation of staging and stratified negation [14, 15]. 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 `w` and `w2` to take place at the same **X10** place, but we are forced to use two different places in order to refer negatively to the `edge` relation when defining the `noedge` relation.

The introduction discussed the basic features of the **L10** language. 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 still a work in progress; it will be available from `http://l10.hyperkind.org` when released. 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 [2, 6, 9, 18]. 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. **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 $\texttt{nat}$) 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)`
$l : \text{goto } l'$	`line L (goto L')`
$l : \text{if } (x \mathbin{?} c) \text{ goto } l'$	`line L (if X Cmp C L')`
$l : \text{return } x$	`line L (return X)`

We capture instructions as a type $\texttt{inst}$, declared on line 6 in Figure 1. **L10** has three built-in types: $\texttt{string}$, the type of string constants, $\texttt{nat}$, the type of non-negative integers, and $\texttt{t}$, 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 $\texttt{w0}$. There are three relevant relations here, at least initially:

- succ $l\ l'$: line l has (potential) successor l' in the program CFG.
- def $l\ x$: line l *defines* variable x.
- 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 rules are logically represented in informal notation as follows:

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

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.

```
1    // Commands
2
3    w0: world.
4    line: nat -> inst -> rel @ w0.
5
6    inst: type.
7    binop:  t -> t -> t -> t -> inst.
8    move:   t -> t -> inst.
9    loadc:  t -> t -> inst.
10   goto:   nat -> inst.
11   if:     t -> t -> t -> nat -> inst.
12   return: t -> inst.
13
14   // Extracting relevant information
15
16   succ: nat -> nat -> rel @ w0.
17   def:  nat -> t -> rel @ w0.
18   use:  nat -> t -> rel @ w0.
19
20   line L (binop X Y Op Z) ->
21       succ L (L+1),
22       def L X,
23       use L Y, use L Z.
24
25   line L (move X Y) ->
26       succ L (L+1),
27       def L X,
28       use L Y.
29
30   line L (loadc X C) ->
31       succ L (L+1),
32       def L X.
33       // no variables used
34
35   line L (goto L') ->
36       succ L L'.
37       // no variables defined
38       // no variables used
39
40   line L (if X Cmp C1 L') ->
41       succ L L', succ L (L+1),
42       // no variables defined
43       use L X.
44
45   line L (return X) ->
46       // no successors
47       // no variables defined
48       use L X.
```

Figure 1. Program analysis: capturing relevant information from the program.

```
50   // Liveness analysis
51
52   wLive: world.
53   live: nat -> t -> rel @ wLive.
54
55   use L X -> live L X.
56
57   live L' U,
58   succ L L',
59   not (def L U) ->
60       live L U.
```

Figure 2. Program analysis: liveness.

```
62    // Constructing the interference graph
63
64    wInter: world.
65    interferes: t -> t -> rel @ wInter.
66
67    def L X,
68    not (line L (move X _)),
69    succ L L',
70    live L' Z,
71    X != Z ->
72        interferes X Z.
73
74    line L (move X Y),
75    succ L L',
76    live L' Z,
77    X != Z,
78    Y != Z ->
79        interferes X Z.
```

Figure 3. Program analysis: interference.

2.1.2 Liveness analysis

With `succ`, `def`, and `use` defined, we can now implement liveness analysis. Liveness involves the introduction of one new relation:

- live l x: at line l, variable x is *live*.

Usually, liveness analysis is presented in the form of data flow equations for which we compute a least fixed point. Here, we run the rules to saturation, which can also be seen as a least fixed point computation.[1] The **L10** program for liveness, shown in Figure 2, transcribes an informal description of liveness: a variable is live wherever it is used and, if a variable x is live at line l', it is live at all the predecessors of l' that do not, themselves, define x.

The second rule, on lines 57-60 in Figure 2, makes it clear that this is 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. Note that we are forced to put liveness analysis at a different **L10** world because we refer negatively to the definition of def l x.

2.1.3 Constructing the interference graph

Interference graphs are used for register allocation, and we can now compute the interference graph from the liveness relation. The vertices of the interference graph are the variables of the low-level language. There should be an edge between two vertices x and y if it is necessary to assign different machine registers to x and y. Again, one new relation is involved:

- interferes x y: the variables x and y interfere and cannot be assigned to the same register.

We compute the interference graph by observing that an instruction that defines a variable x interferes with any variable *different* from x that is live in a successor line. The move instruction presents an exception, because a move instruction $l : x \leftarrow y$ does not force us to assign x and y to different registers: if the two registers are the same, the move is simply redundant. The **L10** code for computing the interference graph is shown in Figure 3.

Again, the negations are properly stratified: the world `w0` that the `line` relation is associated with can be staged before world `wInter`, and inequality and equality of terms is a primitive for all types in **L10**. Unlike liveness, interference itself is not recursive.

---

[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.

```
81    // Necessary variables
82
83    nec: nat -> t -> rel @ w0.
84    line L (if X Comp C L1) -> nec L X.
85    line L (return X) -> nec L X.
86
87    // Neededness analysis
88
89    wNeed: world.
90    needed: nat -> t -> rel @ wNeed.
91
92    nec L X -> needed L X.
93
94    needed L' X,
95    succ L L',
96    not (def L X) ->
97        needed L X.
98
99    use L Y,
100   def L X,
101   succ L L',
102   needed L' X ->
103       needed L Y.
```

Figure 4. Program analysis: neededness.

```
105   // Dead-code analysis
106
107   wDead: world.
108   dead: nat -> rel @ wDead.
109
110   def L X,
111   succ L L',
112   not (needed L' X) ->
113       dead L.
```

Figure 5. Program analysis: dead code.

2.1.4 Neededness analysis

So far, the **L10** program we have been developing has used worlds for stratified negation, but we have not explored any opportunities for parallelism. To illustrate the generality this approach and additional opportunities for parallelism, we will now consider a neededness analysis, which will inform dead-code elimination. The liveness information computed for register allocation is not exactly appropriate, 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:

- nec l x: at line l, x is necessary for control flow or as the return value

- needed l x: at line l, x is needed

The first relation is defined at world `w0` like the def, succ, and use relations. The second relation is seeded by these necessary variables and propagated backwards, similar to liveness analysis. We define the relation `needed` at world `wNeed`, noting that since this world does not depend on `wLive`, both analyses can be staged in parallel. The **L10** code for neededness analysis is given in Figure 4.

```
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 6. Regular expression matching.

2.1.5 Dead-code elimination

Having performed a neededness analysis, identifying dead code, showing in Figure 5, is straightforward: code is dead if it defines a variable that is not needed. We introduce one final relation:

- dead l: the command at line l is dead code

As we have mentioned, neededness and liveness analysis are independent as presented above and can proceed in parallel. However, if we wanted to take into account that no interference can arise from dead code, we could add a premise not(dead L) for the rules computing interference. The soundness of this rule then depends on the fact that no actual instructions will be emitted for dead code. In this case liveness and neededness can still be computed in parallel, but the interference graph will require both computations (we will have to stage wDead before wInter).

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

```
35   db1 = (token "f" 0, token "o" 1,
36          token "o" 2, token "EOF" 3)
37      @ w1 (seq (tok f) (some (tok o))).
38
39   db2 = (token "b" 0, token "o" 1,
40          token "o" 2, token "EOF" 3)
41      @ w1 (seq (tok f) (some (tok o))).
```

Figure 7. Regular expression querying.

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. As in the introduction, 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 14 of Figure 6 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 16 of Figure 6. 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.

The rules that define the match relation, lines 17-33 in Figure 6, 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 18); given an arbitrary token in a position i, we can always match the empty string in that position (line 20); 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

As discussed in the introduction, 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 can our regular expression

```
43    // Adding negation to regular expressions
44
45    neg:  regexp -> regexp.
46
47    token _ I,
48    token _ J,
49    I <= J,
50    not (match RE I J) ->
51       match (neg RE) I J.
52
53    db3 = (token "d" 0, token "a" 1,
54           token "a" 2, token "EOF" 3)
55      @ w1 (seq (neg (alt (tok "b") (tok "c")))
56             (some (tok a))).
57
58    db4 = (token "b" 0, token "a" 1,
59           token "a" 2, token "EOF" 3)
60      @ w1 (seq (neg (alt (tok "b") (tok "c")))
61             (some (tok a))).
```

Figure 8. Regular expressions with negation, and two queries that try to match the strings "daa" (db3) and "baa" (db4) against the regular expression $\neg(b \mid c)(a+)$.

matcher reach saturation, considering 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 program like the one in Figure 6, because the rules dealing with alternation $(r_1 \mid r_2)$ on lines 31 and 33 of Figure 6 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 6 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, but only 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 parallel evaluation,
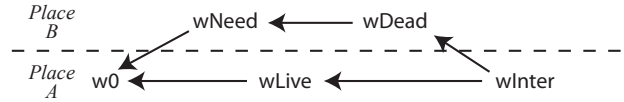

Figure 9. Place assignment for the program from Section 2.1

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 47-51 of Figure 8, 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 [12].

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. The latter is currently used for parsing; ultimately we anticipate replacing the Standard ML parser with a parser written in **L10**, so that the language will be written entirely in **L10** and **X10**. The interpreter will be available from http://l10.hyperkind.org when it is released.

Within a particular stage – that is, when performing computation at a particular world – the interpreter is not fundamentally different than a 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) [9].

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 made the change suggested in Section 2.1 to make interference analysis at world `wInter` dependent on the dead code analysis at world `wDead`, then a query of the form `db = (...)@wInter` will schedule computation on two different places as long as more than one place is available. One way of scheduling this query is shown in Figure 9, where the arrows between worlds represent dependencies.

The result of this breadth first search is a task list for every **X10** place. In the case of Figure 9, place A will derive the `def`, `succ`, `use`, and `nec` relations (at world `w0`), then will derive liveness information (at world `wLive`), and then will block until the dead code analysis is finished at place B. Once the dead code analysis
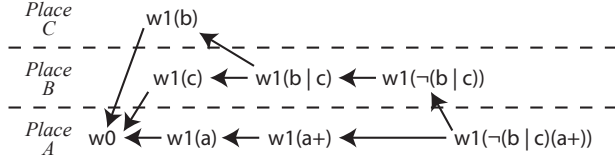
```
Place
 C        w1(b)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Place
 B        w1(c)  ←  w1(b | c)  ←  w1(¬(b | c))
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Place
 A    w0  ←  w1(a)  ←  w1(a+)  ←─────────  w1(¬(b | c)(a+))
```

Figure 10. Place assignment for the queries in Figure 8

finishes, it will use that information to compute the interference graph (at world `wInter`). Simultaneously, the process at place B will immediately block until the basic relations are derived at place A; when this information has been derived, the process at place B computes neededness analysis (at world `wNeed`) and then the interference analysis (at world `wInter`).

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 1) all subterms have been considered or else 2) the amount of unique branches exceeds the available parallelism. In the second case, once we have at least n different subterms of the original term distributed among n different places, then all of the subsequent subterms of those terms will be staged at the same world.

As a concrete example, the regular expression queries from Figure 8 can be scheduled as shown in Figure 10 if there are at least three places available. On the other hand, if there are only two unique places, then it will definitely be the case that `w1(b)` and `w1(c)` are staged at the same **X10** place, as b and c are both subterms of (`alt b c`).

This assignment is interesting in part because it is effectively the kind of search performed by a *backward-chaining* interpreter for logic programs in the style of Prolog. A consequence of this strategy is that programs have to be structured in such a way that we can effectively perform this search: if we have some third **L10** world named `w2` in the regular expression example and a one-place relation `matchN` at `w2`, then we *cannot* write the rule "`match RE 0 N -> matchN N.`" The world `w2` is potentially dependent on `w1(RE)` for *every* regular expression RE, but because there are countably many of these worlds, we cannot perform static scheduling for all of them at query time.

The example above is a bit contrived, but there are more realistic programs that are precluded by the requirement of static scheduling; relaxing this constraint is an important direction for future work.

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 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++ [4] and a McAllester-style interpreter that exposes an API to Standard ML [17].

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.

4.1 Program transformations to optimize communication

Because it can be comparatively costly to transmit data from one place to another in a language like **X10** with a partitioned global address space (PGAS), 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. However, this is not always the optimal behavior. Consider the following code from the interference analysis (lines 67-72 in Figure 3):

```
def L X,
not (line L (move X _)),
succ L L',
live L' Z,
X != Z ->
   interferes X Z.
```

It is entirely possible that the **X10** place assigned to world `w0` (associated with the relations `def`, `line`, and `succ`) will not be the same as the **X10** place assigned to world `wInter` for the `interferes` relation. If the former is at place A and the latter is at place B, the evaluation of this rule will begin the following steps:

- Transmit every fact `def L X` from place A to place B.

- For each such L and X, transmit whether or not there exists a `Y` such that `line L (move X Y)` from place A to place B.

- For each remaining L, transmit all the L' such that `succ L L'` from place A to place B.

- ...

The result, in other words, is a large amount of potentially unnecessary communication between the two **X10** worlds. In particular, the variable L is not used at all in the last two premises or the conclusion. It seems much better to transmit from place A to place B all pairs of variables X and L' such that, for some L, `def L X`, `not (line L (move X _))`, and `succ L L'` hold simultaneously. 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.

4.2 Foundations in constructive provability logic

The theoretical basis for **L10** is intended to be constructive provability logic [15], an intuitionistic modal logic that allows stratified negation to be modeled as regular intuitionistic negation. Constructive provability logic allows the logic to analyze the proofs of propositions at accessible worlds, which is consistent with our discussion of the relationship between **L10** worlds and stratified 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.

A final theoretical issue is that, while we only used negation in this paper, we would sometimes like to use more general *aggregates*. In the program analysis example, aggregates would allow us to find the maximum line number L where X is live, or the total number of variables that interfere with Y, or the list of all line numbers containing conditional jumps. We have used constructive provability logic as the starting point for a proof-theoretic justification for stratified negation in deductive databases; we hope to be able to use the same framework to explain the operation of aggregates.

There are several additional ways we are interested in using explicit worlds to increase the expressiveness of **L10** programs. For instance, it may be possible to use worlds in the setting of constructive provability logic to model **LDL++**'s choice operators and XY-stratification [1] or to express preferences for some facts or rules over others when making arbitrary choices [5].

4.3 Distributed worlds

A final observation is that, in this paper, we have only really 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.

Allowing worlds to be indexed by string keys that are *not* known in advance 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**. The key produced by the map function, a string, would be the index to a world, and thus would get sent to one of the available places based on the hash of the string, at which point an aggregate could be used to collect all keys and apply the reduce function to the result.

Furthermore, if negation was not involved, it would not necessarily be problematic to allow worlds indexed by some key to refer to instances of the same world indexed by a different key. This would be valuable in the program analysis example: if the line relation describing the program, as well as the def, use, and line relations, could be distributed between multiple **X10** places, it might allow the program analysis and similar programs to utilize more parallelism than is exposed when we use non-indexed worlds.

Acknowledgments

References

[1] F. Arni, K. Ong, S. Tsur, H. Wang, and C. Zaniolo. The Deductive Database System LDL++. *Theory and Practice of Logic Programming*, 3(1):61–94, 2003.

[2] 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.

[3] 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.

[4] 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.

[5] H. Ganzinger and D. A. McAllester. Logical algorithms. In *Proceedings of the 18th International Conference on Logic Programming*, pages 209–223, 2002.

[6] 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.

[7] Y. A. Liu and S. D. Stoller. From datalog rules to efficient programs with time and space guarantees. *ACM Transactions on Programming Languages and Systems*, 31(6):21:1–21:38, 2009.

[8] 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.

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

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

[11] 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.

[12] T. C. Przymusinski. *On the declarative semantics of deductive databases and logic programs*. Morgan Kaufmann Publishers Inc., 1988.

[13] 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.

[14] 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.

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

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

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

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

[19] 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.