Live Logic Programming

Spencer Rugaber, Zef Hemel, Kurt Stirewalt
LogicBlox, Inc.
Atlanta, GA USA
{spencer.rugaber, zef.hemel, kurt.stirewalt}@logicblox.com

***Abstract*—Logic programming languages are today used to build applications accessing large database systems. This raises the possibility of building live development environments for them. Of particular interest is how specific language features such as level of abstraction, transactions, etc. affect the design of such an environment. In this paper, we explore this question for a specific logic language, Datalog, contrast traditional and live approaches for its tooling and discuss issues that arise.**

***Index Terms*—Logic programming, Datalog, Live programming environment, Intentionality.**

I. Live Programming

Can live programming environments be used with logic programming languages? To answer this question, we must first define what we mean by a *live programming environment*. For the purposes of this paper, an Interactive Development Environment (IDE) is *live* to that extent that the distance, both temporal and intentional, between making a change to a program and seeing its effect is small.

Temporal distance measures the time lag between making the change and seeing the results. It can be caused by delays due to compilation, execution, network latency, database access, etc. Intentional distance is also important. A software developer, when making a change to a program is thinking at a certain level of abstraction, using a specific vocabulary of concepts and with a particular intent. For the *intentional distance* to be small, the response given by the development environment should be at the same level of abstraction, use the same vocabulary and provide information useful in determining whether the intent has been satisfied. For example, imagine a situation in which a programmer has changed the name of a variable in a declaration as the first step in globally renaming it. Here, the programmer is thinking at the global level of abstraction, substituting a name that better reflects the program's requirements, and which thereby improves the program's readability. A live IDE that informs the programmer of all of the temporarily invalid uses of the old variable name has established a significant intentional distance between the programmer's goal and the feedback given. More intentional alternatives include a high-level global-rename refactoring operation or a delay in providing feedback until the macro edit has been completed.

We would like to better understand how the features of a programming language effect the temporal and intentional distance with which it can be handled by a live IDEs. In particular, the research challenge for designers of such environments is to tune the feedback provided, both in its content and in its frequency, so that at each moment in time, developers are provided actionable information tailored to the specific goals that they are trying to achieve. It is the purpose of this document to explore factors that affect such feedback for logic programming languages, specifically Datalog.

II. Logic Programming

The goal of logic programming languages is the direct expression of program requirements/specifications in program code thereby eliminating the need to provide implementation details. This approach is sometimes also called *declarative programming*. The most prominent example of a logic programming language is Prolog [1], which has now been in use for forty years. To make logic programming viable in Prolog, however, certain non-declarative constructs, such as cut, were added enabling the programmer to control the interpreter's search of its goal tree. Instead of Prolog, we consider the more purely declarative logic programming language Datalog, which invented by the database community as an alternative to SQL.

Datalog's power arises from its ability to efficiently deal with large amounts of data while avoiding many of the low-level implementation details found in other languages. The basic unit of data representation in a Datalog program is the *predicate*, which is a named, typed collection of facts of fixed arity. Predicates may describe entities (representatives of objects of interest in the world the program is modeling) usually with a corresponding reference scheme, properties of entities, and relationships among entities. For example, an employment database may have employee entities, each referenced by an employee identification number, the birthdate property, and the supervisory relationships among employees.

There are two categories of predicates in Datalog—EDBs and IDBs. An EDB predicate, which is stored in the *extensional database*, comprises a set of asserted facts, usually obtained from an external data source, such as an input file. An IDB is stored in the *intensional database* and consists of a set of facts derived using rules, which are made up of a *head* and a *body*. If the body evaluates to true with respect to the current set of known facts, then a new IDB fact is derived as specified by the head. Using the terminology of relational databases and SQL, such rules can be used to specify views resulting from table selections, joins and projections. Further, because rules can be recursive, more powerful programs can be written than in standard SQL. In addition to logical operators, such as conjunction and negation, Datalog rules may also make use of arithmetic and aggregation operators, such as TOTAL and MAX. Additionally, Datalog programs may contain *constraints*, which are purely declarative statements about database state. For example, a constraint may indicate that a particular binary predicate is strictly functional in nature.

Because Datalog programs control large, possibly distributed, databases their evaluation is broken into atomic units called *transactions*, each of which has two stages. The *initial* stage is used for processing queries and for on-demand evaluation of EDB assertions. During the *final* stage, IDB predicates are updated by continually interpreting all active program rules until a fixed point is reached; that is, until no further changes occur. If at any time a constraint is violated, the current transaction aborts and the contents of the database revert to its state before the transaction began. When a non-aborting transaction completes, it is said to *commit*.

A typical Datalog program might begin by inputting a data file into a database to which rules are applied to derive new facts that can be later queried. The program itself might include a schema in the form of a set of predicate declarations and accompanying constraints. Rules are specified to compute the required results. Result queries may also be prepared for later presentation via output files or direct display to users.

III. Feedback

Given the above description of Datalog and Datalog programming, questions arise as to what feedback a development environment can give to the programmer and when to give it. For example, what effect does the transactional nature of Datalog execution have on information delivery? If the program intends a group of changes to be effected by a single transaction, then providing feedback with small temporal latency may not be of much use to the programmer because of its large intentional distance. Table I. illustrates some of the kinds of feedback that might be provided by a live logic programming environment, when it would be appropriate to provide them and what actions the environment might take to help the developer deal with the situation.

IV. Observations

We note several differences between the kinds of feedback provided in traditional and a logic programming languages and how they might affect a live development environment. First, of course, is the commit process. The nature of logic programs is such that immediate feedback of computed results when making one of a set of interdependent editing change might increase intentional distance. It is only when a set of related changes has been completed that the computed results reflect the programmer's complete intentions.

Logic programs normally comprise a large set of small (one to four line) rule specifications. Each rule has its own arguments and local variables, thus acting more like a function than a statement in an imperative programming language. Interesting semantic errors arise due to the interdependencies among the numerous, loosely coupled rules. Consequently, seeing these dependencies is quite helpful to understanding where a problem might arise. Dependency visualizations can be either static or dynamic. An example of a static dependency between two rules is when an IDB predicate computed in the head of one rule is dependent on one or more predicates in its body. The body predicates may, in turn, be dependent on predicates in its body. The body predicates may, in turn, be dependent on predicates computed in other rules until, eventually, EDB predicates are reached. Moreover, because of recursion, the dependencies need not be treelike and can form a general directed graph.

TABLE I. Feedback to Logic Programmers

Activity	*Situation*	*Action*
Rule specification	Use of undeclared predicate	Generate declaration
Rule specification	Wrong number / type of arguments	Highlight matching predicate declarations; (semi) automatic correction
Rule specification	Duplicate rule head	Highlight matching rules
Rule specification	Unsafe rule	Highlight matching rules
Rule specification	Missing base case for recursive rule	Highlight matching rules
Predicate / rule declaration selection	Program reading	Highlight referenced and/or dependent predicate declarations
Entity declaration	Missing reference scheme	Generate default reference scheme
Property predicate declaration	Missing entity reference	Offer choices
Predicate / constraint declaration	Unknown predicate reference	Offer choices
Constraint specification	Wrong number / type of arguments	Highlight matching predicate declarations; (semi) automatic correction
EDB fact assertion	Undeclared predicate	Offer choices; generate predicate declaration
EDB fact assertion	Wrong number / type of arguments	Offer choices
EDB fact assertion	Constraint violation	Display constraint and violating values; allow correction
EDB modification	Database maintenance	Display altered views
Query	Routine use	Display results
Query	Unknown predicate; wrong number / type of arguments	Offer alternatives
Transaction execution	Commit	Highlighted display of new IDB facts
Transaction submittal	Delayed execution	Performance improvement suggestions
Transaction execution	Commit	Display of provenance
Transaction execution	Constraint violation	Rollback; display of constraint; display of violating data

Dynamic dependencies are even more interesting. The computation of a specific predicate may depend on thousands of other computations. Because of the bottom-up, fixed-point nature of Datalog's evaluation algorithm and its recursive rules, the dependency graph may be cyclic. Fortunately, such cycles can, in most cases, be *stratified* (broken into tiers that are interdependent in only an acyclic way). This suggests the possibility of visualizations that abstract a dynamic dependency graph in a way to make it more comprehensible.

The dynamic dependency graph enables the computation of various kinds of provenance. In general, *provenance* provides a historical record of activities contributing to the ultimate state of a database. Specifically, we can talk about *where, how, why* and *why not* provenance for database records [2, 3]. *Where* provenance tells us about the input sources of designated

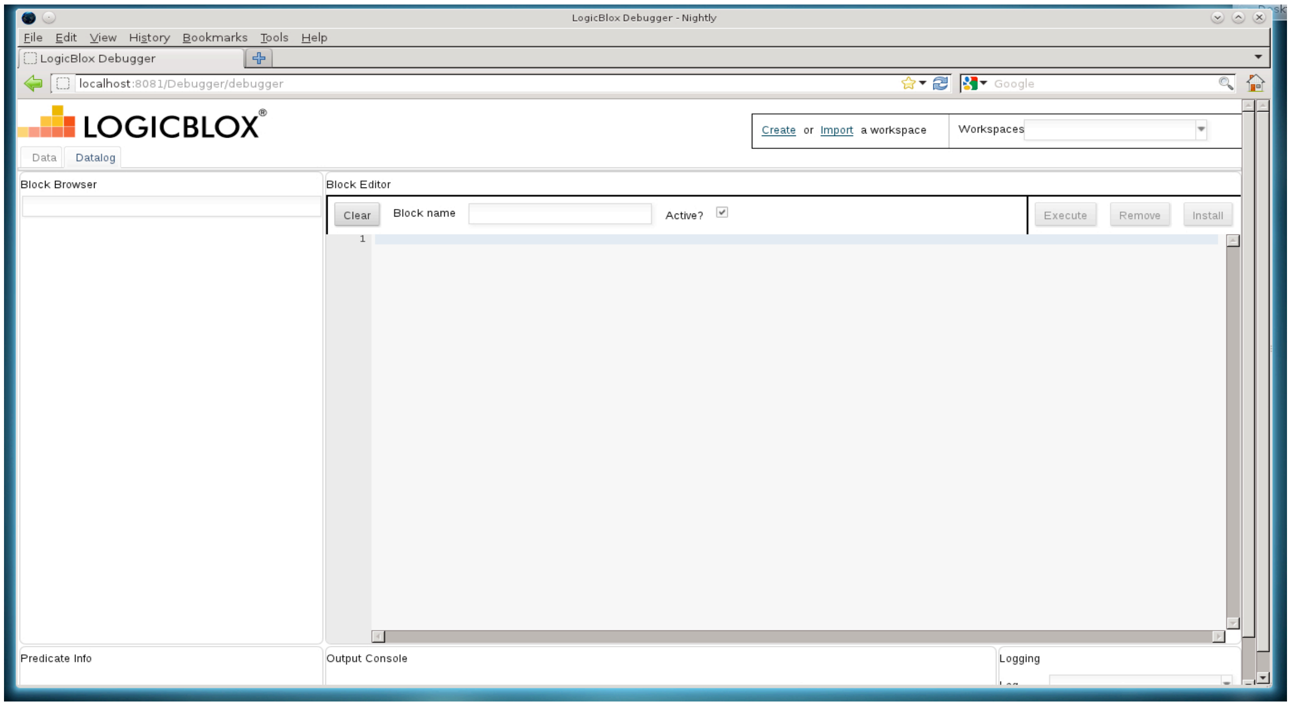

Figure 1: A Traditional IDE for Datalog

output values. In many cases, an output value produced by a query is ultimately copied from some input value, and *where* provenance provides this connection. For example, in debugging, *where* provenance enables us to focus on a particular test case that lead to anomalous output. Note that because Datalog has set-based semantics (in contrast with SQL's multi-set semantics), there may be multiple input values that might have given rise to the output value.

Why provenance provides a more extensive but less precise description of the production of an output fact. In particular, it comprises all sets of input facts that might have contributed to its production. Whereas *where* provenance is concerned with values, *why* provenance is concerned with sets of facts. An example use of *why* provenance is if we wish to understand which input fact influence an output fact in a *what-if* scenario.

More extensive still is *how* provenance, which tells us not only which sets of input facts contribute to the production of an output fact but also which database operations (i.e. rules) were used in that production. Hence *how* provenance enables detailed analysis of the operational steps in a computation. An example of the use of *how* provenance would be to provide a justification for a final result.

Most interesting of all is *why-not* provenance [4]. Here the goal is to help the programmer determine why a given output fact was not produced. Obviously, there may be many reasons, and *why-not* provenance is produced using abductive heuristics. Nevertheless, such suggestions can facilitate program understanding and result justification.

V. A Traditional IDE for Logic Programming

The benefit of a live programming environment is a reduction of the distance from ideation to realization. Such a distance include both time and intent aspects. Having instantaneous response aids the former but at the potential cost of increasing the latter. This increase takes the form of spurious messages and outputs that do not reflect a program state whose meaning is important to the programmer. We claim that a useful live environment should focus on reducing the time distance only for meaningful states. To see how to do this, we contrast traditional and exploratory IDEs for Datalog logic programming.

A traditional IDE for Datalog we have experimented with is shown in Figure 1. The user interface is organized around distinct displays that are customized to support development according to three perspectives: data/spreadsheeting, source code, and diagramming. Each display provides one or more graphical components that target its given perspective plus a number of components that are common across all of the displays. These common components include the output console, the schema browser, and the project selector.

The data display provides a grid that can be used for tasks that require interaction with data. Such tasks include displaying the contents of predicates, executing ad hoc queries, submitting transactions and even modeling an application domain starting from sample data. By default, the IDE opens to this display. The Datalog code display provides a text editor that can be used for tasks that involve the manipulation of large amounts of code. Such tasks include creating and editing programs. The schema display provides graphical tools that support navigating, modeling, and editing the schema of a project or database. The graphical schema language is ORM [5], and a reverse-engineering tool exists to map Datalog code to ORM.

Note that when using the traditional IDE's code view a programmer is required to explicitly switch to the spreadsheet display in order to view the data. Although not obvious in Figure 1, he/she is also required to explicitly press buttons in order to install rules into the IDB or alter facts in the EDB.

VI. A Live IDE for Logic Programming

A live IDE for logic programming needs to provide both less and more than a traditional IDE. On the one hand, the live IDE must avoid giving distracting responses until a transactional unit is available for execution. In the traditional IDE, a transaction unit was explicitly signified by the developer. For a livelier feel, it would help to abstract over the idea of *transactions*; that is, transactions might be optimistically committed, using dependency analysis to determine when such a unit is available and has been changed. If this later subsequently leads to an invalid state, then the transaction is aborted and the database rolled back.

A live interface for logic programming might also provide extensive visualizations of static and dynamic dependency information. Such visualizations would be continuously updated as computations are performed. For example, you could imagine a live environment where on the left you see your logic and on the right you have panels showing predicate data. When you edit your logic on the left, any IDB facts instantly update on the right. In addition, you could edit data on the right, and any other predicate panels whose values are derived from that data could update in a live fashion.

If live updating of (derived) data is supported based on changed logic or changed data, the visualization opportunities increase. For example:

- If you display data in a grid, values that change could temporarily light up in green if they went up, or red if they went down.
- Graphed data could morph from the old graph to the new in an animated way.
- Graphs could be edited by pulling values up and down, thereby updating the underlying data.

VII. Discussion

Brooks has famously distinguished essential and accidental complexity in the context of program development [6]. *Accidental* complexity arises when solving problems is complicated by the tools used to solve them, whereas *essential* complexity is inherent to the problem itself. It should be the goal of any live programming environment to reduce accidental complexity so that the developer can concentrate on essential issues. One way to do so is to make as direct a connection as possible between a proposed solution expressed in code and the results produced by its execution. Directness is enhanced by reducing the temporally and intentional distance between the two.

Of course, all is not smooth sailing. As indicated above, the inherent role of transactions in logic programming compromises the direct connection between program edits and changes to results. Because optimistic commits are inherently heuristic in nature and may have to be rolled back, the programmer will always have to cope with some latency.

Another issue has to do with scale. Databases are usually required for situations where there is lots of data involved. Hence, seeing the effects of a code change will require abstracting results via advanced visualizations. Moreover, the long chains of inferences typical of logic programs mean that provenance will be difficult to present, whether textually of graphically. Similarly, it may be difficult to make sense of constraint violations because, in general, they can act as invariants over the entire state of the database.

A final issue to be concerned with in live logic programming is concurrency. Database systems often support multiple users concurrently viewing and updating database content. If a live development environment is used to update or debug a program accessing a concurrent database, the environment should help manage database volatility. At one extreme, it could create a single-user snapshot of the database. On the other, it might provide monitoring capabilities to warn of potentially faulty situations and recording features so that they can be reproduced.

VIII. Conclusion

Liveness has been part of software development tooling since the time that debugging tools began to allow users to set breakpoints, watch variables, and single-step execution. Multiple, live views of programs and their data have also been with us for at least thirty years. Nevertheless, *dead* environments, in which the temporal and intentional distance between program code and results are large, still prevail in many areas. Fortunately, advances in hardware speed, graphical displays and compiler architectures have enabled many of these early visions to finally be put to practical use.

We began this paper by asking whether live programming environments can be used with logic languages. Because such languages are beginning to be used to develop large-scale applications, they have become attractive candidates for live development environments. However, we determined that certain language features, such as transactions, have to be considered because they affect the temporal and intentional distances of relevance to the programmer. We can now generalize the original question to ask what does *liveliness* mean for different language paradigms? That is, how do we design live environments tailored to specific language features?

Acknowledgment

The authors wish to thank their colleagues at LogicBlox who have designed the version of Datalog and built the platform technology described in this paper.

References

[1] W. F. Clocksin and C. S. Mellish. Programming in Prolog: Using the ISO Standard / 5th Edition. Springer. 2003.

[2] James Cheney, Laura Chiticariu and Wang-Chiew Tan. "Provenance in Databases: Why, How, and Where." *Foundations and Trends in Databases,* 1(4):379–474, 2007.

[3] Grigoris Karvounarakis, Zachary G. Ives, Val Tannen. "Querying Data Provenance. *SIGMOD 2010*, pp. 951-962.

[4] Adriane Chapman and H. V. Jagadish. "Why-Not?" *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data,* 523-534, Providence, Rhode Island, June 29th to July 2, 2009.

[5] Terry Halpin and Tony Morgan. *Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design (2nd ed.).* Morgan Kaufmann, March 2008.

[6] Frederick P. Brooks, Jr. "No Silver Bullet - Essence and Accidents in Software Engineering." *Information Processing 1986, Proceedings of the IFIP Tenth World Computing Conference,* J.-J. Kugler (ed.), 1986, pp. 1069-1076.