Abstract

We demonstrate how a purely declarative language, with the help of strict typing, precise moding, and determinism declarations, can be used to concisely and declaratively express database transactions, including updates. We have begun incorporating transactions into the Aditi deductive database system using an extended form of Mercury as the database programming and query language.

1 Introduction

Ideally, a database programming language should be declarative and should possess a transaction framework within which both queries and database updates can be expressed. In this paper we present language constructs that take us closer to this ideal than previous proposals.

We say a language is able to express queries and updates within a *transaction framework* if it allows a programmer to declare sets of actions that are to be treated as one atomic operation in a multi-user concurrent access database system.

We define a language to be *declarative* if it satisfies two criteria: (1) a programmer can express *what* is to be done with a minimum amount of information on *how* the system is to do it; and (2) it has its foundations in a formally defined mathematical language such as predicate calculus. If you strip SQL of its update facilities, then it can be claimed to meet the criteria of being a declarative language. However, the update features of SQL are not declarative. Most deductive databases also have declarative query languages, but non-declarative update facilities.

There are several reasons why it is desirable for a language (database or otherwise) to be declarative. Firstly, the programmer's task is simplified by not requiring details of *how* tasks are to be carried out. It allows the programmer to write code that is clearer and hence more maintainable. It also has the effect that much of the work of finding efficient execution strategies is left to the compiler. This is especially important for database systems where the best execution strategy may be dependent on the contents of the database which not only might be unknown at the time of code development, but may also vary over time.

The requirement that the language has its foundations in a formally defined mathematical language can simplify the task of developing powerful optimization techniques. Many program transformations used for optimization purposes in logic programming languages would have been more difficult to develop if the languages had not been based on predicate calculus. Indeed, they are almost impossible to apply to those logic programming languages, such as Prolog, that have extra features not based on predicate calculus such as *assert* and *retract*.

Previous attempts to meet the aims of making a language both declarative and able to express database transactions have been unsatisfactory. Our success relies on the use of a logic programming language that is able to express imperative update in a purely declarative way through the use of *type, mode, determinism*, and *uniqueness* information. Our motivation is a desire to incorporate such a declarative language as the query and update language of the next release of the Aditi deductive database system [22], which is currently under development.

An aim of the Aditi project is to produce a database system with programming facilities that are more general than those provided by relational database systems, while not compromising its performance. At its base, Aditi utilizes all of the features of relational technology in a client-server architecture.

The current release of Aditi has only limited support for concurrency and multiple users, because it has no support for transactions at the language level. In the next release of Aditi, we are using Mercury [19] as both the database programming and query language because it is a purely declarative logic programming language with strong typing and strong modes. As it is a general purpose logic programming language, it was not designed with the requirements of a database transaction processing system in mind. However, we will show that a database transaction framework can be added naturally to Mercury. We have completed a prototype implementation of the transaction predicate discussed in this paper.

Previous attempts at incorporating database updates into logic programming languages have needed to either: take the predicate logic on which the language is based and twist it beyond recognition; invent a whole new logic; or develop hybrid solutions that combine an imperative language with a logic programming language. We have found none of these approaches entirely satisfactory. A detailed discussion is presented in Section 5.

The remainder of this paper is organized as follows. Section 2 gives a motivating example. Section 3 reviews the Mercury type, mode, and determinism system, and describes how this system allows input/output to be incorporated into a purely declarative logic programming language. Section 4 introduces our approach to transactions and updates. Section 5 discusses related work. Section 6 discusses future work.

2 Motivating example

The simple case of deleting a single record from a relation is sufficient to introduce the major contribution of this paper. Consider a database transaction to remove account number 1234 from a relation of account balances in a database. This transaction can be expressed as shown in Figure 1 by using Mercury extended with transaction facilities. The terminology and notation used in Figure 1 is explained in subsequent sections.

The predicate `db__transaction` is a library predicate whose first argument `removeAccount(account(1234))` is a call to the `removeAccount` predicate; second argument `Result` is a result value; third argument `S0` represents the state of the world (including the database) before the transaction; and whose last argument `S1` represents the state of the world after the transaction. The semantics of `db__transaction` are described in Section 4.

The transaction goal extracts the current database state from the state of the world and passes it to the predicate `removeAccount`. This in turn has been defined to change the state of the database from a state `D0` before the record deletion, to a state `D` after the record deletion. Delete, insert, and modify predicates are discussed further in Section 4.

3 An overview of Mercury

Mercury is a purely declarative logic programming language designed at The University of Melbourne. Space limitations do not allow us to describe most of the details of the language, so we will limit our description to an overview in this section, and more detailed descriptions where necessary in the remainder of the paper. We refer the reader to [19] for more detail.

Mercury is intended for the construction of large software systems and is thus quite different to other logic programming languages. Many aspects of the language make it a good base language for developing deductive database applications. Following is a brief summary of the features which are important for this paper. Although Mercury syntax is similar to that of Prolog, Mercury requires a number of aspects of the intended behaviour of programs to be declared.

Module systems are important parts of modern languages for enabling groups of programmers to work on the same program with a minimum of interference with each other and for managing the complexity of large projects. Mercury has a module system somewhat like that of Ada. It is particularly useful for database applications because it helps to break down the entire database into more manageable units, and imposes some hierarchy on the relations. In our examples, the prefix `io__` denotes a predicate which is defined in the I/O library, which is a standard part of the system. Similarly, the prefix `db__` denotes a predicate which is defined in our proposed database transaction library.

Mercury is a strongly typed language whose type system is quite similar to that of ML and the Mycroft-O'Keefe type system for Prolog[16]. It requires that types be defined, and that predicates have a declaration that describes the intended type of each argument. Strong typing has long been recognized as a valuable mechanism for making software more reliable and more maintainable, both in the database community and in the general programming community. The line

```
:- pred main(io__state, io__state).
```

declares the predicate `main` with two arguments — both of the type `io__state` which is an abstract type exported by the I/O module. The following code declares a new type `db__result` to which we will refer in Section 4.1.

```
:- type db__result --->
     ok ; exception(string).
```

This type has two functors: `ok` which has no arguments, and `exception` which has a single argument whose type is `string`.

Mercury programs are strongly moded. This means that each predicate has one or more declarations that describe which arguments are inputs and which are outputs. The compiler checks that there is an ordering of the literals in the clause that satisfies the constraints of the mode declarations and reports an error if there is not. The mode information is a superset of the *bound/free* adornments [21] used in many deductive database systems that play an important role in many database rule transformation techniques, such as magic sets (for example, see [1]).

The modes can also specify that an argument should be *unique*, that is that there should be only one live reference to the data object passed in that argument. Uniqueness annotations can be used to ensure that an object is globally unique, and provide a basis for declarative I/O as discussed below.

The mode declaration for `removeAccount` is:

```
:- mode removeAccount(in, di, uo).
```

This indicates that the first argument is an input and should be ground at the time of the call. A mode of `out` would indicate that the argument should be free at the time of the call and will be bound by the predicate. The modes `di` and `uo` are the unique modes *destructive input* and *unique output* respectively. Destructive input means that the argument should be ground and unique at the

```
:- pred main(io__state, io__state).
:- mode main(di, uo) is det.

main(S0, S) :-
    db__transaction(removeAccount(account(1234)), Result, S0, S1),
    ( Result = ok ->
        io__write_string("Transaction succeeded", S1, S)
    ;
        io__write_string("Transaction failed", S1, S)
    ).

:- pred removeAccount(accountNumber, db__state, db__state).
:- mode removeAccount(in, di, uo) is det.

removeAccount(Account, D0, D) :-
    accountBalance_delete(Account, D0, D).
```

Figure 1: Program to add interest to a bank account.

time of the call and may not be used after the call. Unique output means that the argument should be free at the time of the call, and the called predicate will bind it and guarantee that it is unique. One further mode that will be used later in the paper is `ui` or *unique input* which means that the argument is ground and unique at the time of the call, and the called procedure must not cause it to become shared.

Determinism annotations are associated with each mode of a predicate to describe whether the particular mode of that predicate may fail, and whether it may have more than one solution. The determinism information is important to a database because knowledge of the determinism of the goals in a query allows various query optimizations to be performed. Most predicates in applications programs are *deterministic*; they do not fail, and have only a single solution. Predicates that have at most one solution, but may fail are *semi-deterministic*. Predicates that may fail or produce multiple solutions are *nondeterministic*, and predicates that produce at least one solution but may produce multiple solutions are *multi-solutioned*. As determinism annotations, these are abbreviated `det`, `semidet`, `nondet` and `multi`. The two predicates shown in Figure 1 are both declared to be `det`.

Unlike other practical logic programming languages, Mercury contains *no* non-logical constructs such as pruning operators, and even I/O is done in a purely declarative way. Since a pure language is not dependent on a particular execution strategy, they are very useful for programming deductive databases. Many program transformation and optimization techniques (such as constraint propagation, and deforestation, see [8, 10, 11, 20] for examples) can be applied without changing the observed operational semantics. Such transformations can result in dramatic performance improvements.

While Mercury is based on first order predicate logic, it supports higher-order programming. The type and mode systems contain the necessary components to allow programs to create higher-order terms, pass them around and then evaluate them in a disciplined way. The `db__transaction` predicate defined in Section 4.1 demonstrates the syntax used for passing higher-order terms.

Type, mode, and determinism declarations provide a framework in which we can model I/O. I/O is modeled in a purely declarative way by representing the state of the external world as a *unique* object that is destructively updated; predicates that perform I/O take an object representing the state of the world before the I/O is performed and return an object that represents the state of the world after the I/O has been performed. The uniqueness of these structures is enforced by the compiler which ensures that a state of the world cannot be referred to again after it has been destructively updated.

All of these features have several important implications, for a database programming language as well as for general purpose programming. The compiler can check the consistency of the program clauses and the declarations. By doing so, the compiler is able to detect a very large proportion of program errors statically. The compiler has sufficient static information about the behaviour of the program that it can produce very efficient code.

4 Database transactions

The approach Mercury takes to I/O can be extended to database updates within transactions. A database is a shared resource that can be viewed and updated concurrently by more than one user. By treating the database as part of the state of the world, we can give a declarative semantics to database transactions. Database updates are regarded as I/O that affects the state of the world.

For the purposes of this paper, database applications are divided into a *database* consisting of a set of database relations in a relational database, and a set of rules referred to as the *program*. Currently, we only allow updates to the database. Finding a sensible approach to rule updates remains an area requiring further work to solve various semantic and implementation issues.

We extend Mercury in two important ways: by adding a transaction predicate; and by adding predicates to update (insert, delete and modify) each database relation.

4.1 The transaction predicate

Our transaction predicate provides a declarative semantics for a transaction which has the required ACID properties[1] of atomicity, consistency, isolation and durability [6]. It is defined as follows:

```
:- pred db__transaction(
    pred(db__state, db__state),
    db__result, io__state, io__state
    ).
:- mode db__transaction(
    pred(di, uo) is det, out, di, uo
    ) is det.
```

The goal `db__transaction(P, Result, S0, S1)`, takes one state of the world (`S0`), containing the current database state, and transforms it into another state of the world (`S1`), containing a different database state. If the `Result` (whose type was defined in Section 3) is `ok`, then all of the actions defined by `P` were successfully applied to the database state in the initial state of the world and are reflected in the database state in the subsequent state of the world. If the `Result` is `exception`, then none of the actions defined by `P` on the database state in the initial state of the world are reflected in the database state in the subsequent state of the world, and the argument of the `exception` functor will contain an error message indicating how the transaction failed.[2] That is, `P` is applied *atomically* to the initial database state.

If the updates specified by the transaction cause integrity constraint violations, then the transaction will make no changes to the database state, and will return a constraint violation message in `Result`. This ensures that the database remains *consistent*.

[1]The term "ACID" was first used in [7].

[2]In practice, a more sophisticated `db__result` is used — one that distinguishes between different modes of failure such as integrity constraint violations, user requested aborts, deadlocks, crashes, and so on.

`db__transaction` is a deterministic predicate containing `io__state` arguments moded as `di` and `uo`. This ensures that no back-tracking can occur across or into a transaction. The transaction is applied precisely once, and any changes made to the database are reflected permanently. That is, once a transaction is completed, the changes made to the underlying database are *durable*.

A predicate implementing a change to the `io__state` effectively defines a set of differences between one state of the world and another. However, it does not define all of the differences between one state of the world and another. This allows a transaction to be *isolated* from all other transactions while still supporting transaction concurrency and serializability.

To the programmer, the thread of execution appears to be as portrayed in Figure 2. Our transaction is defined to extend from the first `db__state` to the last `db__state`. Other transactions executed by, for example, other users are seen as being executed between the first `io__state` and first `db__state`, and between the last `db__state` and the second `io__state`. These regions are represented by the diagonal arrows in Figure 2. That is, other transactions are seen as changing the state of the world as we change the state of the world, but not as changing the database state as we change the database state.

Any predicates called by `db__transaction` can be safely aborted and the transaction rolled back. This is because predicates called by `db__transaction` can not have `io__state` arguments, and so the only state of the world changes that can occur during a transaction are database state changes. Therefore, all I/O that occurs during a transaction is fully under the control of the database system.

As `db__transaction` requires an `io__state` input argument and the predicate it calls is not provided with an `io__state` argument, it is not possible to nest calls to `db__transaction` inside each other. That is, at the language level we support the classical "flat" transaction model introduced in [5]. We discuss supporting other transaction models in Section 6.

In the examples in this paper, the only values returned by a transaction are a `db__result` value and a new database state. In practice, it is necessary for a transaction to return values resulting from queries. This is achieved by adding an extra output argument to the `db__transaction` predicate and using a *type variable* in the type declaration of `db__transaction` to ensure that it is the same type as that returned by the higher-order term. See [9] for more details.

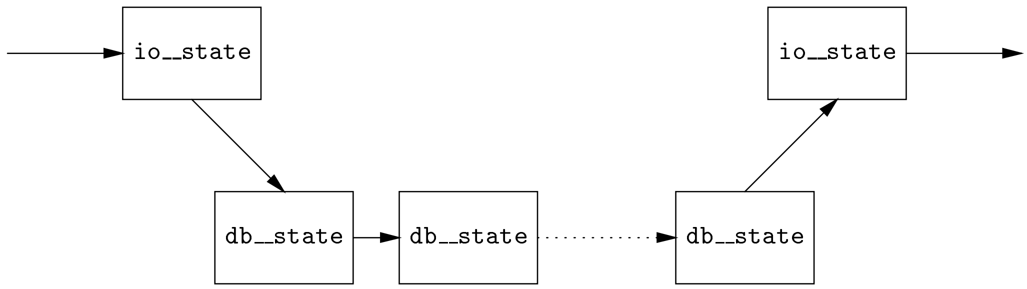

Figure 2: I/O and database state graph for a transaction.

4.2 Update predicates

For each relation in the database, there is a predicate of the same name, for retrieving tuples from that relation. The predicate has one more attribute than the relation. The extra attribute represents the database state. For example, an arity 2 database relation called `edge`, both of whose arguments are strings, will result in an arity 3 predicate `edge(string, string, db__state)`.

Rules referring to a database relation, such as `edge`, must supply a database state. This is the database state which a query on a relation is evaluated against. As the only way to obtain an initial database state is through a `db__transaction` goal, it is guaranteed that the database can only be accessed from within transactions.

Rules for computing the transitive closure of `edge` would be expressed as shown in Figure 3. The database state argument forces the programmer to state when in a sequence of updates the transitive closure is to be computed.

The program given in Figure 4 can be used to insert the edge `edge("b","c")`, and then delete from the `edge` relation all the edges involving nodes that can be reached from node `"a"` *after* the tuple `edge("b","c")` is inserted.

Once the `edge("b","c")` tuple is inserted, no other transaction can read or update that tuple and commit until this transaction finishes and commits.

In the rule for predicate `example`, the first update changes the state of the database from `D0` to `D1`, and the second update finds the nodes reachable from node `"a"` with respect to the updated state of the world `D1`, and performs a deletion to give `D`. The underlying database implementation ensures the atomicity of the transaction.

For each database relation such as `edge`, the database system provides six update predicates: `edge_insert`, `edge_delete`, `edge_modify`, `edge_bulk_insert`, `edge_bulk_delete`, and `edge_bulk_modify`.

The declaration of `edge_insert` is:

```
:- pred edge_insert(
    string, string, db__state, db__state
    ).
:- mode edge_insert(in, in, di, uo)
    is det.
```

The `edge_insert` predicate takes four arguments: the first two are the attributes of the tuple to be inserted into the edge relation, and the remaining two are the before and after states of the database. This is used for inserting a *single* tuple.

The declaration of `edge_bulk_insert` is:

```
:- pred edge_bulk_insert(
    pred(string, string, db__state),
    db__state, db__state
    ).
:- mode edge_bulk_insert(
    pred(out, out, ui) is nondet, di, uo
    ) is det.
```

The `edge_bulk_insert` predicate is used for inserting a set of tuples in one operation. The set is not represented explicitly, but instead is defined using a higher-order term. It is possible to define `edge_bulk_insert` in terms of `edge_insert` and vice versa. There are two reasons why we have chosen to make them two separate primitives. Firstly, there is the issue of convenience: one is more convenient to use than the other in some situations, while in other situations it is the other way around. Secondly, there is the issue of efficiency: implementing one in terms of the other can lead to a serious speed degradation.

The `edge_bulk_insert` predicate has three arguments, the first of which is a higher-order term. This higher-order term must define a predicate that has as many attributes as the relation being inserted into (2 in the case of `edge`), plus one more attribute corresponding to the database state attribute. Note that, conceptually, the database does not change while the higher-order term is evaluated, and so the higher-order term only takes one database state. The database attribute is moded as `ui`, and its other attributes are moded as `out`. A goal such as `edge_bulk_insert(Edges, D1, D)` determines all the values for which `Edges` is true in database state `D1`, and inserts the corresponding records into the `edge` relation yielding database state `D`.

```
:- pred connected(string, string, db__state).
:- mode connected(out, out, ui) is nondet.
connected(X,Y,D) :- edge(X,Y,D).
connected(X,Y,D) :- edge(X,Z,D), connected(Z,Y,D).
```

Figure 3: Transitive closure with database arguments

```
:- pred main(io__state, io__state).
:- mode main(di, uo) is det.
main(S0, S) :-
        db__transaction(example, Result, S0, S1),
        (Result = ok ->
                io__write_string("Transaction OK\n", S1, S)
        ;
                io__write_string("Transaction Failed\n", S1, S)
        ).

:- pred example(db__state, db__state).
:- mode example(di, uo) is det.
example(D0, D) :-
        edge_insert("b", "c", D0, D1),
        edge_bulk_delete(q, D1, D).

:- pred q(string, string, db__state).
:- mode q(out, out, ui) is nondet.
q(X, Y, D) :-
        edge(X,Y,D), (connected("a",X,D) ; connected("a",Y,D)).
```

Figure 4: Program to insert `edge(b,c)` and then delete edges involving nodes that can be reached from node `a`.

The delete and modify predicates assume every relation has an explicitly defined primary key. To delete a record, it is only necessary to provide values for the primary key attributes (alternatively, the entire record could be supplied).

The declaration of edge_bulk_delete is:

```
:- pred edge_bulk_delete(
    pred(string, string, db__state),
    db__state, db__state
    ).
:- mode edge_bulk_delete(
    pred(out, out, ui) is nondet, di, uo
    ) is det.
```

Note that the primary key of the edge relation consists of both of its attributes. We use edge_bulk_delete in Figure 4. Its higher-order term is simply the predicate q. For the given database D1, edge_bulk_delete(q, D1, D) determines all the values for which q(X, Y, D1) is true and deletes the (X,Y) pairs from edge. The result of deleting these zero or more tuples transforms the database from state D1 to state D. Note that, as answers to q rely on the contents of the edge relation, it is important that the edge tuples are not deleted until q is fully computed.

There are situations where answers to the goal used to compute the required updates will be unaffected by the updates themselves. In such situations it will be possible to perform the updates as answers to the query become available before the query has completed.

To support the ability to specify a set of record modifications, we need to avoid situations such as: more than one modification specified for the same record; or one modification changing a record into one for which another modification is defined. We avoid this by restricting modifications to non-primary key attributes, and allowing at most one modification for each instance of the primary-key. Refer to [9] for more details on the modification predicates.

Our approach to database updates can be extended to allow for hypothetical queries. These are queries computed with respect to a database that is modified temporarily. This could be modeled by writing rules that make updates, then compute answers to queries with respect to the new database state, but return the original database state argument. Our current scheme does not allow this due to the destructive-in and unique-out moding of the database state arguments. This problem can be solved by defining a higher-order predicate called, for example, db__hypothetical, that takes a higher-order term that performs updates and returns answers in much the same way as db__transaction. The difference would be that db__hypothetical would return the original database state.

4.3 Discussion

Current database systems require the code of a transaction to be written in two dissimilar languages. In the case of relational systems, the part that queries the database system is written in SQL while the code that decides what updates to apply is written in a imperative "host" language such as C. Even deductive database systems such as LDL [3], CORAL [18], SDS [12] and Glue-Nail [4] make this distinction. For example, in the Glue-Nail system, the Nail declarative query language is complemented by the Glue imperative update language.

We have shown that this rigid separation is not necessary; our approach allows all of the code of the transaction to be written in one language. We believe this makes the development of transactions significantly simpler, for at least two reasons. First, the use of one language reduces or eliminates the "conceptual impedance mismatch" that exists between the set-at-a-time query language and the (usually) tuple-at-a-time host language. Second, since the code is declarative, it can be executed either by the database system (which will use set-at-a-time algorithms) or by the host logic programming system (which will use tuple-at-a-time algorithms). We will initially allow the programmer to designate which predicates should be executed by which mechanism. We also intend to investigate algorithms that the system might use to make this choice automatically and (hopefully) optimally.

Having a single unified language that expresses queries and updates as well as the rest of the program allows the system to analyze and optimize the whole program without artificial boundaries between its various components. For example, it allows information that comes from a part of the program that would be written in the host language in a conventional system to be used to optimize the execution of queries or updates, and vice versa.

Having mathematical logic as the basis of this unified language is advantageous because it does not specify an order of evaluation. The implementation is free to reorder computations any way it likes as long as it respects the dataflow constraints implied by the mode and uniqueness declarations: if operation A needs some data that is computed by operation B, then operation B must be done first, while if operations A and B both read some data but operation B requires that data to be unique input (because it wants to update that data item destructively) then operation A must be done first. Implementations can use this freedom to maximize the amount of work they do on each traversal of the relations involved in their operations. This will minimize the number of traversals, which will in turn minimize I/O costs and/or the amount of memory required.

5 Related Work

Various attempts have been made to incorporate database updates into logic programming languages. We have found that all previous attempts have been unsatisfactory.

Most Prolog systems support the *assert* and *retract* primitives for adding and deleting database facts. Indeed, adding and deleting of Prolog rules at runtime is often supported. The semantics of assert and retract assume a fixed execution model in which Prolog goals are proven in a fixed order determined by the ordering in which rules are written and the ordering of body literals.

At best, assert and retract hinder compiler optimizations as it can become very difficult, if not impossible, for a compiler to determine when rule ordering and body literal ordering can be safely changed to enhance performance. Not only do assert and retract make compiler optimizations difficult, but their semantics can become very unintuitive, especially in the presence of back-tracking. Indeed, the semantics varies from one implementation to another [14].

Attempts have been made at devising more declarative semantics for updates in logic programming.

Some have approached the problem by using alternative logics such as *modal temporal logic* (e.g. [23]), and *transaction logic* [2]. These are valid approaches. However, these logics are more complex than the simpler and more familiar predicate calculus. Since our approach is based on predicate calculus, we believe that it is more natural to express transactions in our language than these other approaches.

Other attempts to work within the framework of predicate calculus have only partially succeeded. For example, a transaction primitive is used for transforming database states in [17]. Under this proposal, until a transaction commits, database updates are not visible even to the process performing the updates. Our work demonstrates that this restriction is unnecessary. Another drawback of [17] is that there is no mechanism to stop a program from back-tracking over a committed transaction (the semantics of which is unclear). This is avoided in our proposal by insisting that transactions only occur in deterministic rules containing the unique state of the world variables.

A more recent approach that also attempts to stay within the framework of predicate calculus is given in [24]. Under this approach, the sequence of database state changes resulting from a sequence of database updates is captured by giving all predicates an extra argument representing the sequence number. By insisting that rules that contain updates have a variable N as the sequence argument in the body literals, and a term $s(N)$ as the sequence argument in the rule head, a form of local stratification through updates is ensured. This approach does not suffer from the semantic problems that others do suffer from, and indeed may be a worthwhile alternative. However, we believe that our approach allows a more natural expression of transactions.

6 Future work

In Section 4 we noted that our current `db__transaction` predicate provides a flat transaction model to the programmer at the language level. One area of future work is to support a more general transaction model.

The underlying database engine can be implemented using any compatible transaction model. For example, to facilitate high performance B-tree index updates, the ARIES method [13] uses a nested top-level transaction model. The additional features, over the flat transaction model, are not visible to the programmer, so need not be expressed in the language.

One possible approach is to provide support for the nested transaction model [15] by defining a `db__subtransaction` predicate as follows:

```
:- pred db__subtransaction(
    pred(db__state, db__state),
    db__result, db__state, db__state
    ).

:- mode db__subtransaction(
    pred(di, uo) is det, out, di, uo
    ) is det.
```

However, this predicate only supports a limited nested transaction model. It does not allow a programmer to distinguish between those subtransactions which are independent, and so can be run in parallel, and those subtransactions which are dependent on other subtransactions, requiring an execution order to be imposed on them. One possible solution to this problem lies in the area of program analysis, but this requires further investigation.

7 Conclusion

We have argued that the ideal database programming language is declarative. However, a database programming language must allow queries and updates to be expressed within a transaction framework. Previous attempts to design a database programming language which is both declarative and can express transactions have been unsatisfactory.

In this paper, we have shown how transactions and updates guaranteeing the ACID properties can be incorporated naturally into a pure logic programming language that supports strong types, modes, determinism and uniqueness. Our extension is a small one, and its use fits directly into the style of programming encouraged by such languages. It also exploits the ability of the compiler to detect a very large fraction of programmer mistakes based on the information supplied by the programmer's declarations.

The database programming language, based on Mercury, is being incorporated into the second major version of the Aditi deductive database. We already have a prototype implementation of the transaction mechanism discussed in this paper. The Mercury system has been freely available on our World Wide Web site for about two years now.

8 Acknowledgements

The authors were supported by Australian Research Council grants, the Key Centre for Knowledge Based Systems, and the Cooperative Research Centre for Intelligent Decision Systems. The second and fourth authors were also supported by Australian Postgraduate Awards.

References

[1] F. Bancilhon, D. Maier, Y. Sagiv and J.D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth ACM PODS Symposium on Principles of Database Systems*, pages 1–15, 1986.

[2] Anthony J. Bonner and Michael Kifer. Transaction logic programming. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 257 – 279, Budapest, 1993.

[3] Danette Chimenti, Tony O'Hare, Ravi Krishnamurthy, Shamim Naqvi, Shalom Tsur, Carolyn West and Carlo Zaniolo. An overview of the LDL system. *IEEE Data Engineering*, Volume 10, Number 4, pages 52–62, December 1987.

[4] M. A. Derr, S. Morishita and G. Phipps. The Glue-Nail deductive database system. *The VLDB Journal*, Volume 3, Number 2, pages 123–160, 1994.

[5] K. P. Eswaran, J. Gray, R. Lorie and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, Volume 19, Number 11, pages 624–633, 1976.

[6] J. Gray and A. Reuter. *Transaction processing: Concepts and techniques.* Morgan Kaufmann, 1993.

[7] T. Härder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, Volume 15, Number 2, pages 287–317, 1983.

[8] James Harland and Kotagiri Ramamohanarao. Constraint propagation for linear recursive rules. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 683–699, 1993.

[9] David B. Kemp, Thomas Conway, Evan P. Harris, Fergus Henderson, Kotagiri Ramamohanarao and Zoltan Somogyi. Database transactions in a purely declarative logic programming language. Technical report 96/5, Department of Information Systems, The University of Melbourne, 1996.

[10] David B. Kemp, Kotagiri Ramamohanarao, Isaac Balbin and Krishnamurthy Meenakshi. Propagating constraints in recursive deductive databases. In *Proceedings of the First North American Conference on Logic Programming*, pages 981–998, October 1989.

[11] David B. Kemp and Peter J. Stuckey. Analysis based constraint query optimization. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 666–682, 1993.

[12] W. Kießling, H. Schmidt and W. Strauß. DECLARE and SDS: Early efforts to commercialize deductive database technology. *The VLDB Journal*, Volume 3, Number 2, pages 211–244, 1994.

[13] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, Volume 17, Number 1, pages 94–162, March 1992.

[14] C. Moss. Cut and paste – defining the impure primitives of Prolog. In *Proceedings of the Third International Conference on Logic Programming*, pages 686–694, London, 1986.

[15] J. E. B. Moss. *Nested transactions: An approach to reliable distributed computing.* MIT Press, 1985.

[16] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, Volume 23, pages 295–307, 1984.

[17] Lee Naish, James A. Thom and Kotagiri Ramamohanarao. Concurrent database updates in Prolog. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 178 – 195, Melbourne, 1987.

[18] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan and Praveen Seshadri. The CORAL deductive system. *The VLDB Journal*, Volume 3, Number 2, pages 161–210, 1994.

[19] Zoltan Somogyi, Fergus Henderson and Thomas Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, Volume 29, Number 1-3, pages 17–64, 1996.

[20] Divesh Srivastava and Raghu Ramakrishnan. Pushing constraint selections. In *Proceedings of the Eleventh ACM PODS Symposium on Principles of Database Systems*, pages 301–315, 1992.

[21] Jeffrey D. Ullman. Implementation of logical query languages for databases. *ACM Transactions on Database Systems*, Volume 10, Number 3, pages 289–321, 1985.

[22] Jayen Vaghani, Kotagiri Ramamohanarao, David B. Kemp, Zoltan Somogyi, Peter J. Stuckey, Tim S. Leask and James Harland. The Aditi deductive database system. *The VLDB Journal*, Volume 3, Number 2, pages 245–288, 1994.

[23] D. S. Warren. Database updates in pure Prolog. In *Proceedings of the 1984 International Conference on Fifth Generation Computer Systems*, pages 244 – 253, Tokyo, 1984.

[24] Carlo Zaniolo and Resa Sadri. A simple model for active rules and their behaviour in deductive databases. In *The Proceedings of the Second ICLP-Workshop on Deductive Databases*, pages 13–26, 1994.