Persistent Functional Languages: Toward Functional Relational Databases

Lesley Wevers
University of Twente
l.wevers@utwente.nl
Expected Graduation Date: 2016
Supervised by Marieke Huisman & Maurice van Keulen

ABSTRACT

Functional languages provide new approaches to concurrency control, based on techniques such as lazy evaluation and memoization. We have designed and implemented a persistent functional language based on these ideas, which we plan to use for the implementation of a relational database system. With such a database system, we aim to show that lazy evaluation can be used to perform online schema transformations. Additionally, our persistent language allows database programs to be written as stored transactions, a mechanism similar to stored procedures. At a later stage, we want to leverage existing verification tools for the automatic verification of postconditions over these functional transactions.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages - Persistent programming languages; H.2.4 [**Database Management**]: Systems - Transaction processing; D.3.2 [**Programming Languages**]: Language Classifications - Functional Languages

Keywords

Concurrency Control; Lazy Evaluation; Memoization; Online Schema Transformation

1. INTRODUCTION

Integrating programming languages with relational database systems promises the construction of database programs in a single environment. However, despite decades of work on developing such *persistent languages*, they have not been able to gain much traction. Most programming languages do not integrate well with the relational model due to different data models and semantic foundations. Moreover, performing high level optimizations on imperative programs is difficult. In contrast to most early work in this area where imperative languages are used, we explore the use of *functional languages* as the basis for a persistent language. Functional languages are already used for querying XML databases [13] because of the possibility to define high-level optimizations by rewriting queries, and because of the possibility to parallelize query execution without explicit user guidance. However, the use of functional languages for the *updating* of databases has gone largely unexplored.

In functional programming languages, computation is based on the evaluation of expressions. Functions in a *pure* functional language have no side-effects and are deterministic, thus always producing the same output given the same input. This allows the output of a function to be *memoized* to avoid expensive recomputation. Moreover, purity implies that every evaluation order that terminates leads to the same value. For example, this allows lazy evaluation and parallel evaluation of expressions. To facilitate efficient execution, functional programs are commonly represented as graphs. This allows efficient copying of state by sharing common subgraphs, and avoids duplication of suspended computations in copied states. A modern functional programming language that implements most of these ideas is Haskell [9].

Functional languages allow us to optimize transaction execution in ways not possible in current database systems. In particular, functional languages naturally provide the ability to perform operations on databases lazily [12]. Moreover, if a transaction has to be retried, we can use memoization to reuse results of work done in the previous attempt [15]. Additionally, reasoning about functional programs is generally easier than reasoning over imperative programs due to the absence of side-effects, providing a strong basis for the automatic verification and optimization of database programs.

Our main goal is to develop a persistent functional language, and implement a relational database system inside this language. To use functional languages for database systems, Trinder proposes modelling of database updates as functions that take as input the current version of the database, and which return a new version of the database, together with an observable result [12]. Trinder shows that these *functional transactions* can be executed in parallel through lazy evaluation, while trivially satisfying the ACID properties. However, he also shows that there are limitations to this model. To mitigate this, we propose the use of optimistic execution and memoization, and show how to integrate this with the model proposed by Trinder. In Section 2 we review the work by Trinder and discuss our proposal.

While Trinder's model provides the basis for functional transaction processing, it does not provide a suitable programming model for persistent languages. To solve this

SIGMOD'14 PhD Symposium, June 22, 2014, Snowbird, UT, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2924-8/14/06 ...$15.00.
http://dx.doi.org/10.1145/2602622.2602627.

problem, Nikhil proposed that the state consists of a set of value bindings, where transactions can update multiple bindings [8]. In Section 3, we review Nikhil's model, and describe a programming language and runtime system that we are developing based on this model. In contrast to early persistent functional languages, a major goal of our language is to allow parallel execution of transactions.

Our next goal is the development of a relational database system in our language. We are currently at an early stage in this work. A particular challenge is how to denote database updates effectively. We discuss these issues in Section 4. To show the applicability of our system, we want to investigate the use of lazy evaluation to allow *online* schema transformations. We discuss some initial ideas on how we plan to do this in Section 5. As a further goal we want to leverage existing tools for program verification to the new setting of persistent functional languages. We describe our plans for this in Section 6.

2. FUNCTIONAL TRANSACTION PROCESSING

In contrast to existing work which only uses functional languages to query databases, we want to use functional languages to perform *updates* on databases. Central to this is the ability to perform transactions in order to guarantee the ACID properties. Trinder has shown how to do transaction processing in a functional language [12], which we will now briefly review.

In Trinder's model, transactions are executed by a *transaction manager*, of which a simple version is given by the following recursive function:

```
tm : S → [S → S × R] → [R]
tm s (tx:txs) =
        let (ns, r) = tx(s) in r : (tm ns txs)
```

This function takes an initial state `s` of type S and a stream of transaction functions of type $S \to S \times R$. The initial state is passed to the first transaction function `tx` as input, which then produces as output the next state `ns` of type S and an observable result `r` of type R. The transaction manager then recursively calls itself with the next state and the tail of the list of transaction functions `txs` to process the remainder of the transactions. As a result, this function produces a list of transaction results. To implement database systems based on this model, Trinder suggest the use of bulk data types to model the state. The type S of the state is conceptually similar to a database schema.

In this model, the sequential execution of transactions guarantees serializability. However, a limitation of Trinder's model is that transaction functions are required to be *total*, i.e., they must always produce a next state. To abort a transaction, a transaction can return the original state. Consistency of the state can be guaranteed by composing a transaction function with a function that validates the result state, and which aborts if the validation fails.

This simple model is not concerned with communication to the outside world, or persistence. In fact, both can be added around the transaction manager function. First, to handle concurrent requests from the outside world, transactions can be serialized before handing them over to the transaction manager. Second, the functional model greatly simplifies the implementation of durable persistence compared with traditional database systems. To be able to recover a database from an inconsistent state as a result of system failure, traditional database systems need to make a copy of every value being mutated before mutating them. In contrast, to guarantee durability in Trinder's model, a transaction function can be journaled before it is executed. If the system crashes, the journal can be replayed to restore the state. To avoid the journal growing too large, and to speed up the process of restoring the state, a snapshot of the state can be created regularly. This approach has been implemented in the ACID-state library [2] for Haskell.

2.1 Lazy Transaction Execution

So far we have not discussed parallel execution of transactions, which is of prime importance for any realistic database system. In fact, the model discussed so far already allows for parallel execution of transactions, as shown by Trinder.

Instead of immediately computing the next state, a transaction can return a closure that computes the next state. This means that updates to the state can execute in constant time by immediately constructing a closure, and moving the burden of the computation to readers of the state. As the closure that computes the state is pure, readers can work on computing the state in parallel without interfering with each other. This mechanism is effectively implemented by lazy evaluation in functional programming languages.

If all readers are working on computing the same closure, parallel computation time is wasted, as they will all get the same result. In practice, parallel readers can avoid working on the same task using a dynamic task scheduler such as work-stealing [4], which is commonly used to implement parallel evaluation in functional languages. To introduce more concurrency, a state computation can recursively construct more closures in a divide-and-conquer style, allowing readers to work on different tasks. This is trivial to implement over recursive data structures such as trees.

An interesting property of this strategy is that readers will only compute those parts of the state that they actually need. Any updates on parts of the state that are not immediately needed are postponed. This means that a bulk update, such as mapping a function over a tree, does not have to block readers from accessing the state. This can for example be used to perform non-blocking schema transformations, as we discuss in Section 5. However, a limitation of this approach is that it can not parallelize all types of transactions. In particular, this method is limited by data dependencies between transactions. A data dependency is essentially similar to a lock in traditional database systems, as a transaction can not proceed until its immediate dependencies have been computed. However, in contrast to locking, data dependencies can be resolved incrementally by performing the computations. For this strategy to work well, recursive functions that compute the state should return parts of the state before recursing, so that subsequent transactions can proceed.

2.2 Optimistic Transaction Execution

In this section we propose an alternative strategy to parallelize the execution of functional transactions. The basic idea is to execute functional transactions optimistically, and to use memoization to reduce the cost of retries. Memoization has already been investigated in the context of software transactional memory [15]. We show that the same ideas can be used to parallelize transaction execution in an optimistic version of the functional model, and we show how to integrate this strategy with lazy evaluation.

In our approach we assume that there is a mutable pointer that points to the current state. When a transaction wants to execute, it gets the pointer to the current state, and evaluates its transaction function over this state. When evaluation completes and the state pointer still points to the state that it initially got, the transaction *commits* by updating the state pointer to the state that it has computed. If the state pointer changed while evaluating the state, this means that some other transaction changed the state. In this case the transaction has to retry its execution using this new state.

In its basic form, parallel execution in this approach does not lead to any useful work, because only one transaction is able to commit, while all other transactions will fail and have to retry with the new state. However, if we (partially) memoize results from the previous execution attempts, this work may not be wasted, and we get useful parallelism.

This approach can be combined with the lazy evaluation approach in a natural way. A transaction can simply commit an unevaluated state to fall back to the lazy evaluation approach. Additionally, a transaction can selectively evaluate only a part of its result state, and partially rely on lazy evaluation for its execution.

An advantage of optimistic execution is that updates do not block readers of the state. This allows transactions to perform expensive computations such as computing aggregates or checking of constraints. Even for small transactions this approach can be useful if we consider that I/O latency may stall the execution of a transaction. A disadvantage of this approach is that a transaction may never be able to commit because the state keeps changing due to concurrent transactions. This is a subject that we still want to investigate. One idea is to first execute a transaction optimistically to prepare as much work as possible, and if it fails to commit, we execute it lazily while reusing the results obtained in the first attempt.

3. PERSISTENT FUNCTIONAL LANGUAGES

While Trinder's model provides the basis for functional transaction processing, it does not provide a suitable programming model for persistent languages. As the basis for our persistent functional language, we build on the model as proposed by Nikhil [8].

In this model, the state is a set of bindings that map names to expressions in a functional language. These expressions can be data, as well as functions that can be used to operate on that data. Given the previous state, a transaction can construct the next state by atomically adding bindings, updating bindings and removing bindings from the current state. Additionally, a transaction can return an observable result by evaluating an expression in the context of the current state.

3.1 Language

We are developing a language based on Haskell for the definition of functional transactions following Nikhil's model. The idea is that a user can write a transaction in this language, and send it to the runtime system for it to be executed. Our language has a few differences compared with Haskell, which we describe in this section. We demonstrate the features of our language using the following example transaction:

```
length [] = 0
length (x:xs) = 1 + length xs
names' = "bob" : names
result = length names'
```

In this transaction, a function `length` is defined that is local to the transaction. Next, `names` in the next state is assigned the value of the string `"bob"` prepended to the list `names` from the previous state, where the prepend function ':' is used as defined in the previous state. Finally, the transaction queries the number of names in the resulting state, using the function `length` that it has just defined. Note that all binding operations are applied simultaneously, the order of bindings does not matter.

In our language, updating a bindings looks similar to defining a function in Haskell. Writing `x' = 5` specifies that `x` has the value `5` in the next state. Removing a binding `b` from the state is done by writing `remove b`. Additionally, bindings that are local to the transaction may be defined. These bindings only exist while the transaction is executing, and are only visible by the transaction defining them. These bindings are written without a prime, e.g. `x = 5`. There is one special local binding, named `result` which specifies the observable result of the transaction.

In contrast to Haskell, our language distinguishes between variables that refer to the previous state, the next state, and local values. Syntactically, we distinguish these by marking variables that refer to the next state with a prime, e.g. `x'`, while variables that refer to values in the previous state and local values are written without a prime.

Our language also features stored transactions, which are similar to stored procedures. A stored transaction can be parameterized, and consists of a set of binding operations that can refer to the arguments passed to the stored transaction. For example, the following transaction defines a stored transaction that adds a name to a list of names and increments a counter:

```
transaction add_name(name) {
  names' = name : names
  count' = count + 1
}
```

At the moment this mechanism is still very basic. A stored transaction is essentially a function over multiple bindings. In the future we want to investigate if we can make stored transactions composable.

3.2 Runtime System

We have implemented a basic prototype implementation of our persistent language in Java. The system accepts transactions written in our language through an HTTP interface, and returns the observable result of the transaction. Transactions are executed by binding variables to their corresponding values, and then updating the bindings in the state as specified by the transaction. The implementation features a parallel graph reducer that allows lazy evaluation of bindings. Currently, our system implements persistence and durability through journaling and snapshotting as described in Section 2. To avoid the state filling up with too many closures, the system forces the evaluation of all bindings updated by a transaction after it has been executed. For more details about the runtime system, we refer the reader to the Master's thesis by the author [14].

3.3 Typing

We are currently in the process of implementing typing for our language. To facilitate this, the binding model is extended with a type for each binding. Additionally, our language allows type definitions to be stored in the state, similar to the way bindings are defined. Like regular variables, type variables can refer to types in the previous state, the next state, as well as local types. Data types can be redefined by requiring that the old definition is not used anymore, i.e., after executing the transaction, no binding in the state has a type that refers to the old definition. This may require updating all values of the old type to the new type, which currently has to be done by the user. We plan to add a facility to the language to do this automatically for the user given a transformation function. The system as just described is independent of the type system used. Currently, we are using the Hindley-Milner type system with type inference [6], which is a standard type system in functional languages.

3.4 Evaluation Strategies

Currently, our persistent language evaluates all bindings lazily. We also want to allow optimistic execution of transactions, as described in Section 2.2. Our idea is to allow forcing the evaluation of certain (sub-)expressions before updating bindings in the state. If at commit time the data on which the sub-expression has been computed has changed, the transaction has to be retried. This approach allows both lazy evaluation and optimistic execution to be used within a single transaction.

Additionally, we want to investigate evaluation strategies that allow weaker consistency guarantees to improve performance. One idea is to split a transaction into multiple parts such that all parts are executed atomically, but where consistency is only guaranteed for the individual parts. This could for example be used to allow a transaction to efficiently update a balanced tree data structure, where the first part of the transaction updates the values in the tree, while the second part of the transaction balances the tree.

4. FUNCTIONAL RELATIONAL DATABASES

We now discuss modelling of relational databases in our persistent functional language. In particular, we want to investigate how to represent and efficiently update functional relations. This work is currently in a very early stage, but we present some of the issues in this section.

A straightforward method to encode relations is to define a record type, and define a relation as a set of such records. However, for efficient searching we also need indices. One approach to allow indexing is by providing each record with a unique identifier, and by defining indices as sorted (multi-)maps from the indexed value to the record identifiers that correspond to this value. For example, to construct a relation of users that have a username, password and age, with indices on the name and age, we could define the following transaction:

```
data User' = User' { name :: String,
        password :: String, age :: Int }
users' = empty :: Map Int User'
users_by_name' = empty :: Map String Int
users_by_age' = empty :: MultiMap Int Int
```

A problem with this approach is that updates quickly become unwieldy. If we want to update records in a relation, we have to construct a new relation with the updated records, as well as new indices that reflect the updates. This is shown by the following example transaction where we update the age for the user "bob":

```
user_id = get "bob" users_by_name
old_user = get user_id users
new_user =
  old_user { age = (age old_user) + 1 }
users' = put user_id new_user users
users_by_age' =
  put (age new_user, user_id) $
  remove (age old_user, user_id) users_by_age
```

Compared with the equivalent SQL update `UPDATE users SET age = age + 1 WHERE name = 'bob'` this approach is too complicated. However, note that the main problem here is one of notation, and not neccesarily the efficiency of the solution. One limitation of this approach is that updates to indices have to be specified explicitly. Ideally, we would like the system to figure out which indices need to be updated, as is done in existing relational database systems. This is still an area that has not been thoroughly investigated, and to the best of our knowledge there is no academic work in this area. However, there is existing work in the form of two libraries for Haskell, the IxSet [2] and HiggsSet library. In their solution they combine indices and data inside one data structure to ensure consistency between indices and the data. Moreover, they use lenses [7] to avoid `put . modify . get` constructs, as seen in the example above, and instead allow updates on fields in a more direct way.

In addition to exploring existing work, we want to investigate other solutions as well. In contrast to a solution based on libraries, we also have the ability to adapt our language to cater specifically for updates of bindings. One preliminary idea is to create a language around bindings that allows operations on sets of bindings, and allow these operations to be composed as functions.

5. ONLINE SCHEMA TRANSFORMATIONS

One of our goals is to allow schema transformations to be performed while a database system is in use. The need for online schema transformations is evident by the research done at Google to solve this problem in the F1 database [11]. For traditional database systems, there already exist tools such as `pt-online-schema-change` [1]. However, these tools are limited to relatively simple schema changes, and are complicated to use. In the end, these solutions attempt to provide transactional behaviour, but have to do so in a complicated way. Ideally, a DBMS should be able to provide the capability to perform online schema transformations transactionally. In contrast to most previous work, we want to investigate lazy schema transformations. In constrast to the approaches used in F1 and `pt-online-schema-change`, a benefit of lazy schema transformations is that, for many practical cases, a schema transformations can take effect immediately.

Lazy schema transformations have already been studied in object-oriented databases [5], but not yet in a functional database setting. Our idea is to perform schema changes by lazily constructing a new version of the database with the new schema from the old database. Many kinds of schema changes can be encoded in this model, such as changing the columns of a table, or splitting and joining of tables. One complication with this model is that entries in the new schema may get a new identifier, for example when two tables are joined. In this case, foreign keys to this table need to be updated accordingly. We expect that this can also be done lazily. Another issue is the creation of indices on the new relation. Preferably we want to copy indices from the source relation if these are still valid. If a transformation on the indexed values preserves the ordering of the values then it is also possible to transform the source index lazily. We expect that on a technical level it is possible to perform most schema transformations online. However, as our goal is to be able to perform schema transformations in a simple way, we want to investigate methods to denote functional schema transformations in a simple way.

We plan to evaluate our approach by adapting the industry standard TPC-C benchmark [3] to include online schema changes. The TPC-C benchmark specifies a database for a warehouse that is constantly being queried and updated through five different types of transactions. We extend TPC-C by performing different kinds of schema changes, while running the original transaction load. We use this benchmark to investigate whether our approach can actually perform these schema changes without blocking the online transactions performed by TPC-C. Moreover, we want to measure the impact of schema changes on the performance of the system. We are currently in the process of implementing an initial version of this benchmark to evaluate `pt-online-schema-change`.

6. VERIFYING DATABASE SOFTWARE

The last topic that we plan to investigate is the formal verification of programs written in our persistent functional language. In current database systems it is difficult to ensure the correctness of database programs. However, a lot of work has already been done on verification in traditional programming languages. Verification tools have improved a lot over the past few years, and are able to prove more and more cases automatically. Instead of developing new techniques for verification, we want to *leverage* existing verification tools and apply them in the new setting of persistent functional languages. In particular, we want to investigate the automatic verification of postconditions on stored transactions. This can be used to verify that stored transactions respect invariants over the state, without the need for runtime checks [10]. Additionally, we can also use this to verify the functional behaviour of transactions, e.g., showing that a result is not always empty.

7. CONCLUSIONS

Our main goal is the development of a relational database in a persistent functional language, as to integrate a programming language and a database system into a single environment. We have seen that functional languages provide new ways of executing transactions in parallel through lazy evaluation and memoization. To apply these techniques, we are developing a persistent functional language that provides a programming model for the specification of functional transactions. Currently, we are extending this language with a type system and mechanisms to allow the specification of evaluation strategies for functional transactions.

The next step in our project is the modelling of a relational database in our language. The main issue for now seems not to be technical, but notational. While possible, specifying relational database updates in a persistent functional language is currently much more difficult than specifying an update in SQL. To evaluate our system, we want to perform online schema transformations to showcase the applications of functional transaction processing. Further down the road, we plan to leverage existing program verification tools to verify functional database applications.

8. REFERENCES

[1] The pt-online-schema-change manual: http://www.percona.com/doc/percona-toolkit/2.1/pt-online-schema-change.html, April 2014.

[2] The Happstack Book: Modern, Type-Safe Web Development in Haskell: http://www.happstack.com/docs/crashcourse/index.html, April 2014.

[3] TPC-C Benchmark Specification: http://www.tpc.org/tpcc/spec/tpcc_current.pdf, April 2014.

[4] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *J. ACM*, 46(5), Sept. 1999.

[5] Y. Cheung. *Lazy Schema Evolution in Object-Oriented Databases*. PhD thesis, MIT, 2001.

[6] L. Damas and R. Milner. Principal Type-schemes for Functional Programs. In *Proc. of the 9th ACM Symposium on Principles of Programming Languages*, New York, NY, USA, 1982. ACM.

[7] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[8] R. Nikhil. Functional Databases, Functional Languages. In *Proceedings of the First Workshop on Persistent Objects*. Springer-Verlag, 1985.

[9] S. Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, 2003.

[10] D. Plexousakis and J. Mylopoulos. Accommodating Integrity Constraints During Database Design. In *Proceedings of the International Conference on Extending Database Technology*. Springer, 1996.

[11] I. Rae, E. Rollins, J. Shute, S. Sodhi, and R. Vingralek. Online, Asynchronous Schema Change in F1. *Proc. VLDB Endow.*, 6(11), Aug. 2013.

[12] P. Trinder. *A Functional Database*. PhD thesis, University of Oxford, 1989.

[13] P. Wadler. XQuery: A Typed Functional Language for Querying XML. In *Advanced Functional Programming*, Lecture Notes in Computer Science. Springer, 2002.

[14] L. Wevers. A Persistent Functional Language for Concurrent Transaction Processing. Master's thesis, University of Twente, 2012.

[15] L. Ziarek and S. Jagannathan. Memoizing Multi-Threaded Transactions. In *Workshop on Declarative Aspects of Multicore Programming*, 2008.