Persistent Languages

Integrate programming languages and database systems:

- Resolve impedance mismatch
- Optimize database updates
- Simplify verification
Pure Functional Languages

Pure functions always return the same result for the same arguments.

Opportunities for optimization:
- Memoization
- Lazy and parallel evaluation
- Rewriting
Functional Persistent Languages

Functional languages are successfully used for querying databases: e.g. XQuery.

Using functional languages for the updating of databases has not really been explored.
Functional Transactions

transaction : State -> State x Result
Functional Transaction Processing

$tm : \text{State} \times \text{[Transaction]} \rightarrow \text{[Result]}$
Functional States

- Branch "bob"
  - Branch "alice"
    - Leaf "alice": 100
  - Branch "eve"
    - Leaf "bob": 50
    - Leaf "eve": 25
    - Leaf "dan": 25
Functional Updates
Functional Updates

Branch "alice"
  Leaf "alice" 100
  Leaf "bob" 50

Branch "bob"
  Leaf "eve" 50

Branch "eve"
  Leaf "dan" 25
Persistence & Durability

Simple persistence model:

- Journal transactions before executing.
- Regularly snapshot the state.
- Recover state from latest snapshot by replaying journaled transactions from the last snapshot.
Constraints and Aborts

Enforcing constraints over the state:

\[
\text{if check(}next\text{\_state)} \\
\hspace{1em}\text{then (}next\text{\_state, result)} \\
\hspace{1em}\text{else (}prev\text{\_state, Error)}
\]
Transactions

This model satisfies the ACID properties:

- Atomic
- Consistent
- Isolated
- Durable

But how do we execute transactions in parallel?
Concurrent Transaction Processing

```
tm
```

contains a : update : contains b : ...

```
Branch

Branch

Branch

Leaf

Leaf

Leaf

Leaf
```
Concurrent Transaction Processing

- contains a : tm
  - Branch
    - Branch
      - Leaf
      - Leaf
    - Branch
      - Leaf
      - Leaf
    - Branch
      - Leaf
      - Leaf

- update : contains b : ...

Concurrent Transaction Processing

contains a : Nothing : contains b : tm

Branch

map f

Leaf Leaf Leaf Leaf
Concurrent Transaction Processing

contains a : Nothing : contains b : tm

Branch

Branch

Branch

Leaf

Leaf

Leaf

map f    map f

...
Concurrent Transaction Processing

contains a : Nothing : contains b : tm

Branch

Branch

Leaf

Leaf

map f

Leaf

map f

map f

map f

...
Concurrent Transaction Processing

contains a : Nothing : contains b : tm

Branch

Branch

Branch

Leaf

Leaf

Leaf

Leaf

map f

map f

...
Concurrent Transaction Processing

contains a: Nothing: True: tm

Branch → Branch
Branch → Branch
map f → Branch
map f → Leaf
Leaf → Leaf
Leaf → Leaf
Leaf → Leaf
...
Limitations of lazy evaluation

- Concurrency is limited by data dependencies
  e.g. if c then a else b cannot be evaluated until c is evaluated
- Transaction functions must be total
- Memory requirements
Proposed Alternatives

Optimistic concurrency control

- Evaluate (part of) the state before committing
- Use memoization to speed up retries
- Integrates seamlessly with lazy evaluation

Splitting transactions

- Allows for weaker isolation guarantees
Persistent Functional Language

Language derived from Haskell, featuring:

- Functional transaction processing
- Ad-hoc transactions
- Stored transactions

Ongoing work:

- Runtime system
- Type system
- Execution strategies
Modelling Relational Databases

Challenges:
● Automatically using and updating indices
● Automatic optimization
● Exploiting functional execution strategies

Potential solutions:
● Embedded DSL
● Runtime support
Online Schema Transformations

Ongoing work:
- Classifying transformations
- Criteria for online schema transformations
- Benchmark based on TPC-C

We want to perform schema transformations lazily.
Verifying Functional Transactions

Verify postconditions on stored transactions:
- Avoid runtime constraint checks
- Verify functional behaviour

We want to leverage existing tools to this new setting.
Conclusion

Functional persistent languages:

- Integrate programming language and database system
- New techniques for concurrency control
- Many opportunities for optimization

Ongoing and future work:

- Optimistic concurrency control & Memoization
- Modelling relational databases
- Online schema transformations
- Verification of stored transactions
Thank You

Lesley Wevers
lwevers@gmail.com
Persistent Functional Language

{} 
   names’ = [] 
{} { names = [] } 
   names’ = “alice” : names 
{} { names = [“alice”] } 
   result = names
   [“alice”] 
{} { names = [“alice”] }
Persistent Functional Language

```persistent
{ names = ["alice"] }

names’ = “bob” : names

result = names

[“alice”]
```

```persistent
{ names = ["bob", "alice"] }
```
Persistent Functional Language

\{ \text{names} = \[\text{“bob”, “alice”}\] \} 

\text{length’ list} = \text{list match} 

\[\] \rightarrow 0 

(x:xs) \rightarrow 1 + \text{length’ xs} 

\{ \text{names} = \[\text{“bob”, “alice”}\], \text{length} = \lambda \text{list} \rightarrow \ldots \} 

\text{result} = \text{length names} 

2 

\{ \text{names} = \[\text{“bob”, “alice”}\], \text{length} = \lambda \text{list} \rightarrow \ldots \}
Functional Transaction Processing

\[\text{tm} : \text{DB} \times [\text{Transaction}] \rightarrow [\text{Result}]\]

\[\text{tm } s \ (\text{tx:txs}) = \]
\[\quad \text{let } (\text{ns, r}) = \text{tx}(s) \text{ in }\]
\[\quad r : (\text{tm } \text{ns } \text{txs})\]
Concurrent Transaction Execution

Idea: Evaluate states lazily.

update s = (map f s, Nothing)
contains k s = (s, contains k s)

tm (Branch ...) [contains a, update, contains b, ...]
Concurrent Transaction Processing
Future work: Memoization

Remember results of function applications:
- Optimistic execution & retrying transactions
- Aggregate functions:
  - sum, min, max, …
  - constraint checks
- Materialized views
Concurrent Transaction Processing

contains a

Cons

contains b

map f

map f

Cons

Cons

Nothing

Branch

Branch

Leaf

Leaf

Leaf

Leaf

Leaf

Leaf

tm
Concurrent Transaction Processing

contains a

Nothing

Branch

Branch

Branch

Leaf

Leaf

Leaf

map f

map f

contains b

tm

[...]

map f
Concurrent Transaction Processing

- Cons
  - contains a
    - Branch
      - Branch
        - Leaf
        - Leaf
        - Leaf
      - Branch
        - map f
          - map f
            - contains b
              - tm
                - [...]
  - Cons
    - Nothing

Concurrent Transaction Processing

contains a

Nothing

map f

contains b

[...]
Persistent Functional Languages

ACID-State for Haskell implements many of these ideas, however:

- There are no ad-hoc transactions:
  - We can’t do schema changes on the fly
  - We can’t share the state with other programs
- GHC not optimized for this use case:
  - State is limited to main memory
  - Task scheduling not optimized for latency
- ACID-State is limited to lazy evaluation.